



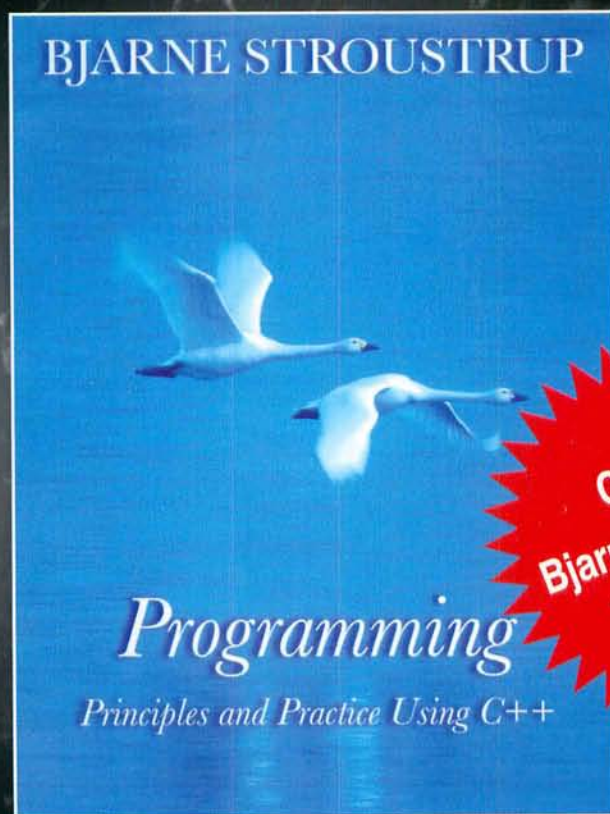
HZ Books
汇智教育

PEARSON

计 算 机 科 学 丛 书

C++程序设计原理与实践

(美) Bjarne Stroustrup 著 王刚 刘晓光 吴英 李涛 译



C++之父
Bjarne Stroustrup
的最新力作

Programming
Principles and Practice Using C++

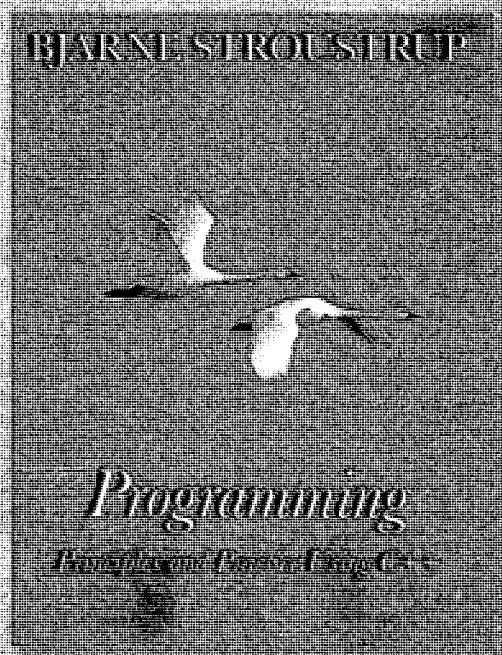


机械工业出版社
China Machine Press

计 算 机 科 学 丛 书

C++程序设计原理与实践

(美) Bjarne Stroustrup 著 王刚 刘晓光 吴英 李涛 译



Programming
Principles and Practice Using C++



机械工业出版社
China Machine Press

www.TopSage.com

本书是 C++ 之父 Bjarne Stroustrup 的最新力作。书中广泛地介绍了程序设计的基本概念和技术,包括类型系统、算术运算、控制结构、错误处理等;介绍了从键盘和文件获取数值和文本数据的方法以及以图形化方式表示数值数据、文本和几何图形;介绍了 C++ 标准库中的容器(如向量、列表、映射)和算法(如排序、查找和内积)的设计和使用。同时还对 C++ 思想和历史进行了详细的讨论,很好地拓宽了读者的视野。

本书语言通俗易懂、实例丰富,可作为大学计算机、电子工程、信息科学等相关专业的教材,也可供相关专业人员参考。

Simplified Chinese edition copyright © 2010 by Pearson Education Asia Limited and China Machine Press.

Original English language title: *Programming: Principles and Practice Using C++* (ISBN 978-0-321-54372-1) by Bjarne Stroustrup, Copyright © 2009.

All rights reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Addison-Wesley.

本书封面贴有 Pearson Education (培生教育出版集团) 激光防伪标签,无标签者不得销售。

封底无防伪标均为盗版

版权所有,侵权必究

本书法律顾问 北京市展达律师事务所

本书版权登记号: 图字: 01-2009-1608

图书在版编目(CIP)数据

C++ 程序设计原理与实践/(美)斯特劳斯特鲁普(Stroustrup, B.)著;王刚等译. —北京:机械工业出版社, 2010.6

(计算机科学丛书)

书名原文: *Programming: Principles and Practice Using C++*

ISBN 978-7-111-30322-0

I. C… II. ①斯… ②王… III. C 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字 (2010) 第 061970 号

机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码 100037)

责任编辑: 李俊竹

北京京北印刷有限公司印刷

2010 年 6 月第 1 版第 1 次印刷

184mm × 260mm · 41.75 印张

标准书号: ISBN 978-7-111-30322-0

定价: 108.00 元

凡购本书,如有缺页、倒页、脱页,由本社发行部调换

客服热线: (010) 88378991; 88361066

购书热线: (010) 68326294; 88379649; 68995259

投稿热线: (010) 88379604

读者信箱: hzsj@hzbook.com

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域中取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自1998年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与Pearson, McGraw-Hill, Elsevier, MIT, John Wiley & Sons, Cengage等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出Andrew S. Tanenbaum, Bjarne Stroustrup, Brian W. Kernighan, Dennis Ritchie, Jim Gray, Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, Abraham Silberschatz, William Stallings, Donald E. Knuth, John L. Hennessy, Larry L. Peterson等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作，而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方式如下：

华章网站：www.hzbook.com

电子邮件：hzjsj@hzbook.com

联系电话：(010) 88379604

联系地址：北京市西城区百万庄南街1号

邮政编码：100037



华章科技图书出版中心

译者序

程序设计是打开计算机世界大门的金钥匙，它使五彩斑斓的软件对你来说不再是“魔术”。C++ 语言则是学习掌握这把金钥匙的有力武器，它优美、高效，从大洋深处到火星表面，从系统核心到高层应用，从掌中的手机到超级计算机，到处都有 C++ 程序的身影。本书适合那些从未有过程序设计经验的初学者，如果你愿意努力学习，本书能帮助你理解使用 C++ 语言进行程序设计的基本原理及大量实践技巧。你所学到的思想，大多数也都可直接用于其他程序设计语言。本书不是初学程序设计语言的简单入门教材，它的目标是能让读者学到基本的实用程序设计技术，因此也可以作为程序设计方面的“第二本书”。基于这样一个目标，注重实践是本书的明显特点。它希望教会你编写真正能被他人所使用的“有用的程序”，而非“玩具程序”。因此，除了基本的 C++ 程序特性之外，本书还介绍了大量的求解实际问题的程序设计技术：如语法分析器程序的设计、图形化程序设计、利用正则表达式处理文本、数值计算程序设计以及嵌入式程序设计等。在其他大多数程序设计入门书籍中，是找不到这些内容的，像调试技术、测试技术等其他程序设计书籍着墨不多的话题，本书也有详细的介绍。程序设计远非遵循语法规则和阅读手册那么简单，而在于理解基本思想、原理和技术，并进行大量实践。本书阐述了这一理念，为读者指引了明确的方向，教会读者如何才能达到编写有用的、优美的程序这一最终目标。

本书的作者 Bjarne Stroustrup 是 C++ 语言的设计者和最初的实现者，也是《The C++ Programming Language》(Addison-Wesley 出版社)一书的作者。 he 现在是德州农工大学计算机科学首席教授，美国国家工程院的会员和 AT&T 院士。在进入学术界之前，他在 AT&T 贝尔实验室工作多年。他是 ISO C++ 标准委员会的创始人之一。本书是他在 C++ 程序设计领域奉献给广大读者的又一经典著作。

本书分为五个部分。第一部分介绍基本的 C++ 程序设计知识，包括第一个“Hello, World!”程序，对象、类型和值，运算，错误处理，函数，类等内容，以及一个计算器程序实例。第二部分介绍输入和输出，首先介绍了输入/输出流的基本概念和格式化输出方法，然后第 12~16 章重点介绍了图形/GUI 类和图形化程序设计。第三部分介绍数据结构和算法，重点介绍了向量、自由内存空间、数组、模板和异常、容器和迭代器以及算法和映射。第四部分希望拓宽读者的视野，介绍了程序设计语言理念和历史、文本处理技术、数值计算、嵌入式程序设计技术及测试技术，此外还较为详细地介绍了 C 语言与 C++ 的异同。第五部分为附录，包括 C++ 语言概要、标准库概要、Visual Studio 简要入门、FLTK 安装以及 GUI 实现等内容。

本书的序、第 0 章、8~11 章、23~25 章、27 章、附录、术语表由王刚翻译，第 4、5、22、26 章由刘晓光翻译，第 1~3 章、17~21 章由吴英翻译，第 6~7 章、12~16 章由李涛翻译。翻译大师经典，难度超乎想象。接受任务之初，诚惶诚恐；翻译过程中，如履薄冰；完成后，忐忑不安。虽然竭尽全力，但肯定还有很多错漏之处，敬请读者批评指正。

译者

2010 年 4 月于南开大学

前言

“该死的鱼雷！全速前进。”

——海军上将 Farragut

程序设计是这样一门艺术，它将问题求解方案描述成计算机可以执行的形式。程序设计中很多工作都花费在寻找求解方案以及对其求精上。通常，只有在真正编写程序求解一个问题的过程中才会对问题本身理解透彻。

本书适合于那些从未有过编程经验但愿意努力学习程序设计的初学者，它能帮助你理解使用 C++ 语言进行程序设计的基本原理并获得实践技巧。我的目标是使你获得足够多的知识和经验，以便能使用最新最好的技术进行简单有用的编程工作。达到这一目标需要多长时间呢？作为大学一年级课程的一部分，你可以在一个学期内完成这本书的学习（假定你有另外四门中等难度的课程）。如果你是自学的话，不要期望能花费更少的时间完成学习（一般来说，每周 15 个小时，共 14 周是合适的学时安排）。

三个月可能看起来是一段很长的时间，但要学习的内容很多，写第一个简单程序之前，就要花费一个小时。而且，所有学习过程都是渐进的：每一章都会介绍一些新的有用的概念，并通过从实际应用中获取的例子来阐述这些概念。随着学习进程的推进，你通过程序代码表达思想的能力——也就是让计算机按你的期望工作的能力，会逐渐稳步地提高。我从不会说：“先学习一个月的理论知识，然后看看你是否能使用这些理论吧。”

为什么要学习程序设计呢？因为计算机文化是建立在软件之上的。如果不理解软件，那么你将退化到只能相信“魔术”的境地，并且将被排除在很多最为有趣、最具经济效益和社会效益的领域之外。当谈论程序设计时，我所想到的是整个计算机程序家族，从带有 GUI（图形用户界面）的个人计算机程序，到工程计算和嵌入式系统控制程序（如数码相机、汽车和手机中的程序），以及文字处理程序等，在很多日常应用和商业应用中都能看到这些程序。程序设计与数学有些相似，如果认真去做的话，它会是一种非常有用的智力训练，可以锻炼我们的思考能力。然而，由于计算机能做出反馈，程序设计又不像大多数数学形式那么抽象，因而对更多人来说更容易接受。可以说，程序设计是一条能够打开你的眼界，将世界变得更美好的途径。最后，程序设计非常有趣。

为什么学习 C++ 这门程序设计语言呢？学习程序设计不可能不借助一门程序设计语言，而 C++ 直接支持现实世界中的软件所使用的那些关键概念和技术。C++ 是使用最为广泛的设计语言之一，其应用领域几乎没有局限。从大洋深处到火星表面，到处都能发现 C++ 程序的身影。C++ 是由一个开放的国际标准组织全面考量、精心设计的。在任何一种计算机平台上都能找到高质量的和免费的 C++ 实现。而且，你用 C++ 所学到的程序设计思想，大多数都可直接用于其他程序设计语言，如 C、C#、Fortran 以及 Java。最后一个原因，我喜欢 C++ 适合编写优美、高效的代码这一特点。

本书不是初学程序设计的简单入门教材，我写此书的用意也不在此。我为本书设定的目标

是:能让你学到基本的实用编程技术的最简单的书籍。这是一个雄心勃勃的目标,因为很多现代软件所依赖的技术,不过才出现短短几年时间。

我的基本假设是,你希望编写供他人使用的程序,并愿意认真负责地、较高质量地完成这个工作;也就是说,我假定你希望达到专业水准。因此,我为本书选择的主题覆盖了开始学习实用编程技术所需要的内容,而不只是那些容易讲授和容易学习的内容。如果某种技术是你做好基本编程工作所需要的,那么本书就会介绍它,同时展示用以支持这种技术的编程思想和语言工具,并提供相应的练习,期望你通过做这些练习来熟悉这种技术。但如果你只了解“玩具程序”,那么你能学到的将远比我所提供的少得多。另一方面,我不会用一些实用性很低的内容来浪费你的时间,本书介绍的内容都是你在实践中几乎肯定会用到的。

如果你只是希望直接使用别人编写的程序,而不想了解其内部原理,也不想亲自向代码中加入重要的内容,那么本书不适合你。请考虑是否采用另一本书或另一种程序设计语言会更好些。如果这大概就是你对程序设计的看法,那么请同时考虑一下你从何得来的这种观点,它真的满足你的需求吗?人们常常低估程序设计的复杂程度和它的重要性。我不愿看到你不喜欢程序设计,只是因为你的需求与我所描述的部分软件之间不匹配。信息技术世界中还有很多部分是不要求程序设计知识的,那些领域可能适合你。本书面向的是那些确实希望编写和理解复杂计算机程序的人。

考虑到本书的结构和注重实践的特点,它也可以作为程序设计方面的第二本书,适合那些已经了解一点 C++ 的人,和那些会用其他语言编程,现在想学习 C++ 的人。如果你属于其中一类,我不好估计你学习这本书要花费多长时间。但我可以给你的建议是,多做练习。因为你在学习中常见的一个问题是习惯用熟悉的、旧的方式编写程序,而不是在适当的地方采用新技术,多做练习会帮助你解决这个问题。如果你曾经按某种更为传统的方式学习过 C++, 那么在进行到第 7 章之前,你会发现一些令你惊奇的和有用的内容。除非你的名字是 Stroustrup, 否则你会发现我在本书中所讨论的内容不是“你父辈的 C++”。

学习程序设计要靠编程实践。在这一点上,程序设计与其他需要实践学习的技能是相似的。你不可能仅仅通过读书就学会游泳、演奏乐器或者开车,你必须进行实践。同样,不读程序、不写程序就不可能学会程序设计。本书给出了大量代码实例,都配有说明文字和图表。你需要通过读这些代码来理解程序设计的思想、概念和原理,并掌握用来表达这些思想、概念和原理的程序设计语言的特性。但有一点很重要,仅仅读代码是不能学会编程实践技巧的。为此,你必须进行编程练习,通过编程工具熟悉编写、编译和运行程序。你需要亲身体验编程中会出现的错误,学习如何修改它们。总之,在学习程序设计的过程中,编写代码的练习是不可替代的。而且,这也是乐趣所在!

另一方面,程序设计远非只是遵循一些语法规则和阅读手册那么简单。本书的重点不在于 C++ 的语法,而在于理解基础思想、原理和技术,这是一名好程序员所必备的。只有设计良好的代码才有机会成为一个正确、可靠和易维护的系统的一部分。而且,“基础”意味着延续性:当现在的程序设计语言和工具演变甚至被取代后,这些基础知识仍会保持其重要性。

那么计算机科学、软件工程、信息技术等又如何呢?它们都属于程序设计范畴吗?当然不是!但程序设计是一门基础性的学科,是所有计算机相关领域的基础,在计算机科学领域占有重要的地位。本书对算法、数据结构、用户接口、数据处理和软件工程等领域的重要概念和技术进行了简要介绍。但本书不能取代对这些领域全面、均衡的学习。

代码可以很有用,同样也可以很优美。本书会帮你了解优美的代码意味着什么,并帮你掌握构造优美代码的原理和实践技巧。祝你学习顺利!

致学生

到目前为止,我在德州农工大学已经用本书的初稿教过 1000 名以上的大一新生,其中 60% 曾经有过编程经历,而剩余 40% 从未见过哪怕一行代码。大多数学生的学习是成功的,所以你也可以成功。

你不一定是在某门课程中来学习本书,我认为本书会广泛用于自学。然而,不管你学习本书是作为课程的一部分还是自学,都要尽量与他人协作。程序设计有一个不好的名声——它是一种个人活动,这是不公正的。大多数人在作为一个有共同目标的团体的一份子时,工作效果更好,学习得更快。与朋友一起学习和讨论问题不是作弊!而是取得进步最有效,同时也是最快乐的途径。如果没有特殊情况的话,与朋友一起工作会使你表达出你的思想,这正是测试你对问题理解和确认你的记忆的最有效的方法。你没有必要独自解决所有编程语言和编程环境中的难题。但是,请不要自欺欺人,不去完成那些简单练习和大量的习题(即使没有老师督促你,你也不应这样做)。记住,程序设计(尤其)是一种实践技能,需要通过实践来掌握。如果你不编写代码(完成每章的若干习题),那么阅读本书就纯粹是一种无意义的理论学习。

大多数学生,特别是那些爱思考的好学生,有时会对自己努力工作是否值得产生疑问。当(不是如果)你产生这样的疑问时,休息一会儿,重新阅读这篇前言,阅读一下第 1 章(“计算机、人和程序设计”)和第 22 章(“思想和历史”)。在那里,我试图阐述我在程序设计中发现了哪些令人兴奋的东西,以及为什么我会认为程序设计是能为世界带来积极贡献的重要工具。如果你对我的教学理念和一般方法有疑问,请阅读第 0 章(“致读者”)。

你可能会对本书的厚度感到担心。本书如此之厚的一部分原因是,我宁愿反复重复一些解释说明或增加一些实例,而不是让你自己到处找这些内容,这应该令你安心。另外一个主要原因是,本书的后半部分是一些参考资料和补充资料,供你想要深入了解程序设计的某个特定领域(如嵌入式系统程序设计、文本分析或数值计算)时查阅。

还有,学习中请耐心些。学习任何一种重要的、有价值的新技能都要花费一些时间,而这是值得的。

致教师

本书不是一门传统的计算机科学的 101 课程,而是一本关于如何构造能实际工作的软件的书。因此本书省略了很多计算机科学系学生按惯例要学习的内容(图灵完全、状态机、离散数学、乔姆斯基文法等)。硬件相关的内容也省略了,因为我假定学生从幼儿园时代就已经通过不同途径使用过计算机了。本书也不准备涉及一些计算机科学领域最重要的主题。本书是关于程序设计的(或者更一般地,是关于如何开发软件的),因此关注的是少量主题的更深入的细节,而不是像传统计算机课程那样讨论很多主题。本书试图只做好一件事,计算机科学不是一门课程可以囊括的。如果本书(本课程)被计算机科学、计算机工程、电子工程(很多我们最早的学生都是电子专业的)、信息科学或者其他相关专业所采用,我希望这门课程能和其他一些课程一起进行,共同形成对计算机科学的完整介绍。

请阅读第 0 章,那里有对我的教学理念、一般教学方法等的介绍。请在教学过程中尝试将这

些观点传达给你的学生。

资源

本书网站的网址为 www.stroustrup.com/Programming, 其中包含了各种使用本书讲授和学习程序设计所需的辅助资料。这些资料可能会随着时间推移不断改进, 但对于初学者, 现在可以找到下面一些资料:

- 基于本书的讲义的幻灯片。
- 一本教师指南。
- 本书中使用的库的头文件和实现。
- 本书中实例的代码。
- 某些习题的解答。
- 可能有用的一些链接。
- 勘误表。

欢迎随时提出对这些资料的改进意见。

致谢

我要特别感谢我已故的同事和联合导师 Lawrence “Pete” Peterson, 很久以前, 在我还未感受到教授初学者的惬意时, 是他鼓励我承担这项工作, 并提供了很多能令课程成功的教学经验。没有他, 这门课程的首次尝试就会失败。他参与了这门课程最初的建设, 本书就是为这门课程所著。他还和我一起反复讲授这门课程, 汲取经验, 不断改进课程和本书。在本书中我使用的“我们”这个字眼, 最初的意思就是指“Pete 和我”。

我要感谢那些直接或间接帮助过我撰写本书的学生、助教以及德州农工大学讲授 ENGR 112 课程的教师, 以及 Walter Daugherty, 他曾讲授过这门课程。还要感谢 Damian Dechev、Tracy Hammond、Arne Tolstrup Madsen、Gabriel Dos Reis、Nicholas Stroustrup、J. C. van Winkel、Greg Versoorder、Ronnie Ward 和 Leor Zolman, 他们对本书初稿提出了一些建设性意见。感谢 Mogens Hansen 为我解释引擎控制软件。感谢 Al Aho、Stephen Edwards、Brian Kernighan 和 Daisy Nguyen, 他们帮助我在夏天躲开那些分心的事来完成本书。

感谢 Addison-Wesley 公司为我安排的审阅人: Richard Enbody、David Gustafson、Ron McCarty 和 K. Narayanaswamy, 他们基于自身讲授 C++ 课程或者大学计算机科学系 101 课程的经验, 对本书提出了宝贵的意见。还要感谢我的编辑 Peter Gordon 为本书提出的很多有价值的意见以及他极大的耐心。我非常感谢 Addison-Wesley 公司为本书组织的制作团队的同仁, 他们为本书的高质量出版做出了很多贡献, 他们是: Julie Grady (校对)、Chris Keane (排版)、Rob Mauhar (插图)、Julie Nahil (制作编辑) 和 Barbara Wood (文字编辑)。

另外, 我本人对本书代码的检查很不系统, Bashar Anabtawi、Yinan Fan 和 Yuriy Solodkyy 使用微软 C++ 7.1 版(2003)和 8.0 版(2005)以及 GCC 3.4.4 版检查了所有代码片段。

我还要感谢 Brian Kernighan 和 Doug McIlroy 为程序设计类书籍的撰写定下了非常高的标准, 以及 Dennis Ritchie 和 Kristen Nygaard 为实用编程语言设计提供的非常有价值的经验。

目 录

第一部分 基本知识

出版者的话

译者序

前言

第0章 致读者	1	第2章 Hello, World!	25
0.1 本书结构	1	2.1 程序	25
0.1.1 一般方法	2	2.2 经典的第一个程序	26
0.1.2 简单练习、习题等	2	2.3 编译	27
0.1.3 进阶学习	3	2.4 链接	29
0.2 讲授和学习本书的方法	4	2.5 编程环境	30
0.2.1 本书内容顺序的安排	6	第3章 对象、类型和值	34
0.2.2 程序设计和程序设计语言	7	3.1 输入	34
0.2.3 可移植性	7	3.2 变量	35
0.3 程序设计和计算机科学	8	3.3 输入和类型	36
0.4 创造性和问题求解	8	3.4 运算和运算符	37
0.5 反馈方法	8	3.5 赋值和初始化	39
0.6 参考文献	8	3.5.1 实例：删除重复单词	41
0.7 作者简介	9	3.6 组合赋值运算符	42
第1章 计算机、人与程序设计	11	3.6.1 实例：重复单词统计	42
1.1 介绍	11	3.7 命名	43
1.2 软件	11	3.8 类型和对象	44
1.3 人	13	3.9 类型安全	45
1.4 计算机科学	15	3.9.1 安全类型转换	46
1.5 计算机已无处不在	15	3.9.2 不安全类型转换	46
1.5.1 有屏幕和没有屏幕	16	第4章 计算	51
1.5.2 船舶	16	4.1 计算	51
1.5.3 电信	17	4.2 目标和工具	52
1.5.4 医疗	18	4.3 表达式	53
1.5.5 信息领域	19	4.3.1 常量表达式	54
1.5.6 一种垂直的视角	20	4.3.2 运算符	55
1.5.7 与C++程序设计有何联系	21	4.3.3 类型转换	56
1.6 程序员的理想	21	4.4 语句	56
		4.4.1 选择语句	57
		4.4.2 循环语句	61
		4.5 函数	64

4.5.1 使用函数的原因	65	6.3.3 实现单词	106
4.5.2 函数声明	66	6.3.4 使用单词	107
4.6 向量	67	6.3.5 重新开始	108
4.6.1 向量空间增长	67	6.4 文法	109
4.6.2 一个数值计算的例子	68	6.4.1 英文文法	112
4.6.3 一个文本处理的例子	70	6.4.2 设计一个文法	113
4.7 语言特性	71	6.5 将文法转换为程序	114
第5章 错误	75	6.5.1 实现文法规则	114
5.1 介绍	75	6.5.2 表达式	115
5.2 错误的来源	76	6.5.3 项	117
5.3 编译时错误	77	6.5.4 基本表达式	118
5.3.1 语法错误	77	6.6 试验第一个版本	119
5.3.2 类型错误	77	6.7 试验第二个版本	122
5.3.3 警告	78	6.8 单词流	123
5.4 连接时错误	78	6.8.1 实现 Token_stream	124
5.5 运行时错误	79	6.8.2 读单词	125
5.5.1 调用者处理错误	80	6.8.3 读数值	126
5.5.2 被调用者处理错误	81	6.9 程序结构	127
5.5.3 报告错误	82	第7章 完成一个程序	131
5.6 异常	83	7.1 介绍	131
5.6.1 错误参数	83	7.2 输入和输出	131
5.6.2 范围错误	84	7.3 错误处理	133
5.6.3 输入错误	85	7.4 处理负数	135
5.6.4 截断错误	87	7.5 模运算: %	136
5.7 逻辑错误	88	7.6 清理代码	138
5.8 估计	89	7.6.1 符号常量	138
5.9 调试	90	7.6.2 使用函数	139
5.9.1 实用调试技术	91	7.6.3 代码格式	140
5.10 前置条件和后置条件	94	7.6.4 注释	141
5.10.1 后置条件	95	7.7 错误恢复	143
5.11 测试	96	7.8 变量	145
第6章 编写一个程序	100	7.8.1 变量和定义	145
6.1 一个问题	100	7.8.2 引入单词 name	148
6.2 对问题的思考	100	7.8.3 预定义名字	150
6.2.1 程序设计的几个阶段	101	7.8.4 我们到达目的地了吗	150
6.2.2 策略	101	第8章 函数相关的技术细节	153
6.3 回到计算器问题	102	8.1 技术细节	153
6.3.1 第一步尝试	103	8.2 声明和定义	154
6.3.2 单词	104	8.2.1 声明的类别	156

第二部分 输入和输出

8.2.2 变量和常量声明	157	第10章 输入/输出流	207
8.2.3 默认初始化	158	10.1 输入和输出	207
8.3 头文件	158	10.2 I/O 流模型	208
8.4 作用域	160	10.3 文件	209
8.5 函数调用和返回	163	10.4 打开文件	210
8.5.1 声明参数和返回类型	163	10.5 读写文件	211
8.5.2 返回一个值	164	10.6 I/O 错误处理	213
8.5.3 传值参数	165	10.7 读取单个值	215
8.5.4 传常量引用参数	166	10.7.1 将程序分解为易管理的 子模块	216
8.5.5 传引用参数	168	10.7.2 将人机对话从函数中分离	218
8.5.6 传值与传引用的对比	169	10.8 用户自定义输出操作符	219
8.5.7 参数检查和转换	171	10.9 用户自定义输入操作符	220
8.5.8 实现函数调用	172	10.10 一个标准的输入循环	220
8.6 求值顺序	175	10.11 读取结构化的文件	222
8.6.1 表达式求值	176	10.11.1 内存表示	222
8.6.2 全局初始化	176	10.11.2 读取结构化的值	224
8.7 名字空间	177	10.11.3 改变表示方法	226
8.7.1 using 声明和 using 指令	178	第11章 定制输入/输出	230
第9章 类相关的技术细节	183	11.1 有规律的和无规律的输入和输出	230
9.1 用户自定义类型	183	11.2 格式化输出	230
9.2 类和成员	184	11.2.1 输出整数	231
9.3 接口和实现	184	11.2.2 输入整数	232
9.4 演化一个类	185	11.2.3 输出浮点数	232
9.4.1 结构和函数	185	11.2.4 精度	233
9.4.2 成员函数和构造函数	187	11.2.5 域	234
9.4.3 保持细节私有性	188	11.3 文件打开和定位	235
9.4.4 定义成员函数	189	11.3.1 文件打开模式	235
9.4.5 引用当前对象	191	11.3.2 二进制文件	236
9.4.6 报告错误	191	11.3.3 在文件中定位	238
9.5 枚举类型	192	11.4 字符串流	238
9.6 运算符重载	193	11.5 面向行的输入	239
9.7 类接口	195	11.6 字符分类	240
9.7.1 参数类型	195	11.7 使用非标准分隔符	241
9.7.2 拷贝	197	11.8 还有很多未讨论的内容	246
9.7.3 默认构造函数	197	第12章 一个显示模型	249
9.7.4 const 成员函数	199	12.1 为什么要使用图形用户界面	249
9.7.5 类成员和“辅助函数”	200		
9.8 Date 类	201		

12.2 一个显示模型	250	14.1.2 操作	289
12.3 第一个例子	250	14.1.3 命名	290
12.4 使用 GUI 库	252	14.1.4 可变性	291
12.5 坐标系	253	14.2 Shape 类	291
12.6 形状	253	14.2.1 一个抽象类	292
12.7 使用形状类	254	14.2.2 访问控制	293
12.7.1 图形头文件和主函数	254	14.2.3 绘制形状	295
12.7.2 一个几乎空白的窗口	255	14.2.4 拷贝和可变性	297
12.7.3 坐标轴	256	14.3 基类和派生类	298
12.7.4 绘制函数图	257	14.3.1 对象布局	299
12.7.5 Polygon	257	14.3.2 类的派生和虚函数定义	300
12.7.6 Rectangle	258	14.3.3 覆盖	301
12.7.7 填充	259	14.3.4 访问	302
12.7.8 文本	259	14.3.5 纯虚函数	302
12.7.9 图片	259	14.4 面向对象程序设计的好处	303
12.7.10 还有很多未讨论的内容	260	第 15 章 绘制函数图和数据图	307
12.8 让图形程序运行起来	261	15.1 介绍	307
12.8.1 源文件	261	15.2 绘制简单函数图	307
第 13 章 图形类	264	15.3 Function 类	309
13.1 图形类概览	264	15.3.1 默认参数	310
13.2 Point 和 Line	265	15.3.2 更多的例子	311
13.3 Lines	267	15.4 Axis 类	311
13.4 Color	268	15.5 近似	313
13.5 Line_style	270	15.6 绘制数据图	316
13.6 Open_polyline	271	15.6.1 读取文件	317
13.7 Closed_polyline	271	15.6.2 一般布局	318
13.8 Polygon	272	15.6.3 数据比例	319
13.9 Rectangle	273	15.6.4 构造数据图	319
13.10 管理未命名对象	276	第 16 章 图形用户界面	324
13.11 Text	277	16.1 用户界面的选择	324
13.12 Circle	278	16.2 “Next”按钮	325
13.13 Ellipse	279	16.3 一个简单的窗口	325
13.14 Marked_polyline	280	16.3.1 回调函数	327
13.15 Marks	281	16.3.2 等待循环	328
13.16 Mark	282	16.4 Button 和其他 Widget	329
13.17 Image	283	16.4.1 Widget	329
第 14 章 设计图形类	288	16.4.2 Button	330
14.1 设计原则	288	16.4.3 In_box 和 Out_box	330
14.1.1 类型	288	16.4.4 Menu	331

16.5 一个实例	332	18.3 必要的操作	374
16.6 控制流的反转	334	18.3.1 显示构造函数	375
16.7 添加菜单	335	18.3.2 调试构造函数与析构函数	376
16.8 调试 GUI 代码	338	18.4 访问向量元素	377
第三部分 数据结构和算法		18.4.1 对 const 对象重载运算符	378
第 17 章 向量和自由空间	343	18.5 数组	379
17.1 介绍	343	18.5.1 指向数组元素的指针	380
17.2 向量的基本知识	344	18.5.2 指针和数组	381
17.3 内存、地址和指针	345	18.5.3 数组初始化	383
17.3.1 运算符 sizeof	347	18.5.4 指针问题	383
17.4 自由空间和指针	347	18.6 实例：回文	385
17.4.1 自由空间分配	348	18.6.1 使用 string 实现回文	386
17.4.2 通过指针访问数据	349	18.6.2 使用数组实现回文	386
17.4.3 指针范围	349	18.6.3 使用指针实现回文	387
17.4.4 初始化	350	第 19 章 向量、模板和异常	391
17.4.5 空指针	351	19.1 问题	391
17.4.6 自由空间释放	351	19.2 改变向量大小	393
17.5 析构函数	353	19.2.1 方法描述	393
17.5.1 生成的析构函数	354	19.2.2 reserve 和 capacity	394
17.5.2 析构函数和自由空间	355	19.2.3 resize	394
17.6 访问向量元素	356	19.2.4 push_back	395
17.7 指向类对象的指针	356	19.2.5 赋值	395
17.8 类型混用：无类型指针和指针 类型转换	357	19.2.6 到现在为止我们设计的 vector 类	397
17.9 指针和引用	359	19.3 模板	397
17.9.1 指针参数和引用参数	359	19.3.1 类型作为模板参数	398
17.9.2 指针、引用和继承	360	19.3.2 泛型编程	399
17.9.3 实例：列表	360	19.3.3 容器和继承	401
17.9.4 列表的操作	362	19.3.4 整数作为模板参数	402
17.9.5 列表的使用	363	19.3.5 模板参数推导	403
17.10 this 指针	364	19.3.6 一般化 vector 类	403
17.10.1 关于 Link 使用的更多讨论	365	19.4 范围检查和异常	405
第 18 章 向量和数组	369	19.4.1 附加讨论：设计上的考虑	406
18.1 介绍	369	19.4.2 使用宏	407
18.2 拷贝	369	19.5 资源和异常	408
18.2.1 拷贝构造函数	370	19.5.1 潜在的资源管理问题	409
18.2.2 拷贝赋值	372	19.5.2 资源获取即初始化	410
18.2.3 拷贝术语	373	19.5.3 保证	411
		19.5.4 auto_ptr	412

19.5.5	vector 类的 RAII	412
第 20 章	容器和迭代器	417
20.1	存储和处理数据	417
20.1.1	处理数据	417
20.1.2	一般化代码	418
20.2	STL 建议	420
20.3	序列和迭代器	423
20.3.1	回到实例	424
20.4	链表	425
20.4.1	列表操作	426
20.4.2	迭代	427
20.5	再次一般化 vector	428
20.6	实例：一个简单的文本编辑器	429
20.6.1	处理行	431
20.6.2	迭代	431
20.7	vector、list 和 string	434
20.7.1	insert 和 erase	435
20.8	调整 vector 类达到 STL 版本 的功能	436
20.9	调整内置数组达到 STL 版本 的功能	438
20.10	容器概览	439
20.10.1	迭代器类别	440
第 21 章	算法和映射	444
21.1	标准库中的算法	444
21.2	最简单的算法：find()	444
21.2.1	一些一般的应用	446
21.3	通用搜索算法：find_if()	447
21.4	函数对象	448
21.4.1	函数对象的抽象视图	449
21.4.2	类成员上的谓词	450
21.5	数值算法	450
21.5.1	累积	451
21.5.2	一般化 accumulate()	452
21.5.3	内积	453
21.5.4	一般化 inner_product()	453
21.6	关联容器	454
21.6.1	映射	454
21.6.2	map 概览	456
21.6.3	另一个 map 实例	458
21.6.4	unordered_map	459
21.6.5	集合	461
21.7	拷贝操作	462
21.7.1	拷贝	462
21.7.2	流迭代器	462
21.7.3	使用集合保持顺序	464
21.7.4	copy_if	464
21.8	排序和搜索	465
第四部分 拓宽视野		
第 22 章	理念和历史	471
22.1	历史、理念和专业水平	471
22.1.1	程序设计语言的目标和哲学	471
22.1.2	编程理念	473
22.1.3	风格/范型	477
22.2	程序设计语言历史概览	479
22.2.1	最早的程序语言	480
22.2.2	现代程序设计语言的起源	481
22.2.3	Algol 家族	485
22.2.4	Simula	490
22.2.5	C	491
22.2.6	C++	493
22.2.7	今天的程序设计语言	495
22.2.8	参考资源	496
第 23 章	文本处理	499
23.1	文本	499
23.2	字符串	499
23.3	I/O 流	502
23.4	映射	503
23.4.1	实现细节	507
23.5	一个问题	508
23.6	正则表达式的思想	510
23.7	用正则表达式进行搜索	511
23.8	正则表达式语法	513
23.8.1	字符和特殊字符	514
23.8.2	字符集	514
23.8.3	重复	515
23.8.4	子模式	516

23.8.5 可选项	516	25.4.3 解决方案: 接口类	561
23.8.6 字符集和范围	516	25.4.4 继承和容器	564
23.8.7 正则表达式错误	518	25.5 位、字节和字	566
23.9 与正则表达式进行模式匹配	519	25.5.1 位和位运算	566
23.10 参考文献	522	25.5.2 bitset	569
第24章 数值计算	525	25.5.3 有符号数和无符号数	570
24.1 介绍	525	25.5.4 位运算	573
24.2 大小、精度和溢出	525	25.5.5 位域	574
24.2.1 数值限制	527	25.5.6 实例: 简单加密	575
24.3 数组	528	25.6 编码规范	579
24.4 C风格的多维数组	528	25.6.1 编码规范应该是怎样的	579
24.5 Matrix库	529	25.6.2 编码原则实例	580
24.5.1 矩阵的维和矩阵访问	530	25.6.3 实际编码规范	584
24.5.2 一维矩阵	532	第26章 测试	589
24.5.3 二维矩阵	534	26.1 我们想要什么	589
24.5.4 矩阵I/O	536	26.1.1 说明	590
24.5.5 三维矩阵	536	26.2 程序正确性证明	590
24.6 实例: 求解线性方程组	537	26.3 测试	590
24.6.1 经典的高斯消去法	538	26.3.1 回归测试	591
24.6.2 选取主元	539	26.3.2 单元测试	591
24.6.3 测试	539	26.3.3 算法和非算法	596
24.7 随机数	540	26.3.4 系统测试	601
24.8 标准数学函数	541	26.3.5 测试类	604
24.9 复数	542	26.3.6 寻找不成立的假设	606
24.10 参考文献	543	26.4 测试方案设计	607
第25章 嵌入式系统程序设计	547	26.5 调试	607
25.1 嵌入式系统	547	26.6 性能	607
25.2 基本概念	549	26.6.1 计时	609
25.2.1 可预测性	551	26.7 参考文献	610
25.2.2 理想	551	第27章 C语言	613
25.2.3 生活在故障中	552	27.1 C和C++: 兄弟	613
25.3 内存管理	553	27.1.1 C/C++兼容性	614
25.3.1 动态内存分配存在的问题	554	27.1.2 C不支持的C++特性	615
25.3.2 动态内存分配的替代方法	556	27.1.3 C标准库	616
25.3.3 存储池实例	557	27.2 函数	617
25.3.4 栈实例	557	27.2.1 不支持函数名重载	617
25.4 地址、指针和数组	558	27.2.2 函数参数类型检查	618
25.4.1 未经检查的类型转换	559	27.2.3 函数定义	619
25.4.2 一个问题: 不正常的接口	559	27.2.4 C++调用C和C调用C++	620

27.2.5 函数指针	621	27.6.2 输入	632
27.3 小的语言差异	622	27.6.3 文件	633
27.3.1 结构标签名字空间	622	27.7 常量和宏	633
27.3.2 关键字	623	27.8 宏	634
27.3.3 定义	623	27.8.1 类函数宏	635
27.3.4 C 风格类型转换	624	27.8.2 语法宏	636
27.3.5 void* 的转换	625	27.8.3 条件编译	636
27.3.6 枚举	626	27.9 实例: 侵入式容器	637
27.3.7 名字空间	626	术语表	644
27.4 动态内存分配	626	参考书目	648
27.5 C 风格字符串	628		
27.5.1 C 风格字符串和 const	629		
27.5.2 字节操作	630		
27.5.3 实例: strcpy()	630		
27.5.4 一个风格问题	630		
27.6 输入/输出: stdio	631		
27.6.1 输出	631		

第五部分 附录[⊖]

附录 A C++ 语言概要
附录 B 标准库概要
附录 C Visual Studio 简要入门教程
附录 D 安装 FLTK
附录 E GUI 实现

⊖ 本书附录部分的内容请读者登录华章网站(www.hzbook.com)下载。——编辑注

“当实际地形与地图不符时，相信实际地形。”

——瑞士军队谚语

本章汇集了多种信息，目的是使你对本书剩余部分的内容有初步了解。你可以略过本章，直接阅读后面你感兴趣的部分。对教师来说，可以立即发现很多有用的内容。如果没有一个好的老师指导你学习本书，请不要试图阅读并理解本章的所有内容，只要阅读“本书结构”一节和“讲授和学习本书的方法”一节的第一部分即可。当你已经能自如编写和执行小程序时，可能需要回过头来重读本章。

0.1 本书结构

本书由四个部分和若干个附录组成：

- 第一部分：基本知识，介绍了程序设计的基本概念和技术，以及开始编写代码需要了解的一些 C++ 语言和库的知识。这部分包括类型系统、算术运算、控制结构、错误处理，以及函数和用户自定义类型的设计、实现和使用等内容。
- 第二部分：输入/输出，介绍了如何从键盘和文件获取数值和文本数据，以及如何生成相应的输出到屏幕和文件。然后介绍了如何以图形化方式表示数值数据、文本和几何图形，以及如何从图形用户界面 (graphical user interface, GUI) 获取输入数据。
- 第三部分：数据结构和算法，关注 C++ 标准库中的容器和算法框架 (标准模板库, standard template library, STL)。展示了容器 (如向量、列表和映射) 是如何 (用指针、数组、动态内存、异常和模板) 实现的以及如何使用它们。还展示了标准库算法 (如排序、查找和内积) 如何设计及使用。
- 第四部分：拓宽视野，通过对 C++ 思想和历史的讨论，通过一些实例 (如矩阵运算、文本处理、测试以及嵌入式系统程序设计)，以及通过 C 语言的一个简单描述，为我们呈现了程序设计的一个全景。
- 第五部分：附录，提供了一些不适合作为教学但很有用的内容，如 C++ 语言和标准库的概要介绍，以及集成开发环境 (integrated development environment, IDE) 和图形用户界面库 (GUI 库) 的入门简介等。

不幸的是，现实世界中的程序设计并不能真正分为完全独立的四个部分。因此，这种划分仅仅是对本书内容的一种粗略分类。我们认为这是一种有用的分类方法 (这是显然的，否则我们不会采用它)，但现实情况往往与这种简洁的分类法相悖。例如，我们很快就会用到输入操作，但对 C++ 标准 I/O 流 (input/output stream, 输入/输出流) 的完整介绍却出现在本书比较靠后的部分。书中有些地方在提出某个概念时需要先介绍另外一些内容，而这与全书的布局不符，对此，我们会在此处简明介绍这些内容，以便更好地提出概念，而不是仅仅指出这些内容的完整介绍在书中什么地方。刻板的分类法更适合于手册而不是教材。

本书内容的顺序是由程序设计技术决定的，而不是程序设计语言特性，参见 0.2 节。附录 A 是按语言特性组织的。

0.1.1 一般方法

在本书中,人称都是直接的,简单清楚,不像很多科技论文中那种惯用的“专业”的婉转称呼方式。当用到“你”时,我们的意思就是“你、读者”;而“我们”指“我们、作者和教师”,或者指读者和我们一起在讨论某个问题,就好像我们在一个教室中一样。

本书的内容组织适合从头到尾一章一章地阅读,当然,你也常常要回过头来对某些内容读上第二遍、第三遍。实际上,这是一种明智的方法,因为当遇到还看不出什么门道的地方时,你通常会快速掠过。对于这种情况,你最终还是会再次回到这个地方。然而,这么做要有个度,因为除了索引和交叉引用之外,对本书其他部分,你随便翻开一页,就开始学习并希望成功,这几乎是不可能的。本书每一节、每一章的内容安排,都假定你已经理解了之前的内容。

本书的每一章都是一个合理的自包含的单元,这意味着应将其一口气读完(当然这只是从理论上讲,实际上由于学生紧密的学习计划,不总是可行的)。这是将内容划分为章的主要标准。其他标准包括:从简单练习和习题的角度,一章是一个合适的单元;每一章提出一些特定的概念、思想或技术。这种标准的多样性使得少数章过长,所以不要教条地遵循“一口气读完”的准则。特别是当你已经考虑了思考题,做了简单练习和一些习题时,通常会发现你需要回过头去重读一些小节和几天前读过的内容。我们按主题将章组合成“部分”,例如,第二部分都是关于输入/输出的内容。每一部分都可以作为一个很好的完整的复习单元。

“它回答了我想到的所有的问题”是对一本教材常见的称赞,这对细节技术问题是理想的,而早期的读者也发现本书有这样的特性。但是,这不是全部的理想,我们希望提出更多初学者可能想不到的问题。我们的目标是,回答那些你在编写供他人使用的高质量软件时需要考虑的问题。学习回答好的(通常也是困难的)问题是学习如何像一个程序员那样思考所必需的。只回答那些简单的、浅显的问题会使你感觉良好,但无助于你成长为一名程序员。

我们努力尊重你的聪明才智,珍惜你的时间。在本书中,我们以专业性而不是精明伶俐为目标,我们宁可节制地表达一个观点而不大肆渲染它。我们尽力不夸大一种程序设计技术或一种语言特性的重要性,但请不要因此低估“这通常是有用的”这种简单陈述的重要程度。如果我们平静地强调某些内容是重要的,我们的意思是,你如果不掌握它,或早或晚都会因此而浪费时间。我们喜欢幽默,但在本书中使用很谨慎。经验表明,人们对什么是幽默的看法大相径庭,不恰当地使用幽默会把人弄糊涂。

我们不会伪称本书中的思想和工具是完美的。实际上没有任何一种工具、库、语言或者技术能够解决程序员所面临的所有难题,至多能帮助你开发、表达你的问题求解方案而已。我们尽量避免“无恶意的谎言”,也就是说,对于那些清晰易理解,但在实际编程和问题求解时容易弄错的内容,对其介绍避免过度简单化。另一方面,本书不是一本参考手册;如果需要C++详细完整的描述,请参考Bjarne Stroustrup的《The C++ Programming Language(Special Edition)》^①一书(Addison-Wesley出版社,2000年)和ISO的C++标准。

0.1.2 简单练习、习题等

程序设计不仅是一种脑力活动,实际动手编写程序是掌握程序设计技巧必不可少的一个环节。本书提供以下两个层次的程序设计练习:

① 本书中文版已由机械工业出版社引进出版,书名为《C++程序设计语言(特别版)》。——编辑注

- 简单练习：简单练习是一种非常简单的习题，其目的是帮助学生掌握一些机械的实际编程技巧。一个简单练习通常由对单个程序的一系列修改练习组成。你应该完成所有简单练习。完成简单练习不需要很强的理解能力、很聪明或者很有创造性。简单练习是本书的基本组成部分，如果你没有完成简单练习，就不能说完成了本书的学习。
- 习题：有些习题比较简单而有些则很难，但大多数习题都是想给学生留下一定的创造和想象的空间。如果时间紧张，你可以做少量习题。但至少应该弄清楚哪些内容对你来说比较困难，在此基础上应该再多做一些，这才是学习成功之道。我们希望本书的习题都是学生能够做出来的，而不是需要超乎常人的智力才能解答的复杂难题。但是，我们还是期望本书习题能给你足够多的挑战，能用光甚至是最好的学生的所有时间。我们不期待你能完成所有习题，但请尽情尝试。

另外，我建议每个学生都能参与到一个小的项目中去（如果时间允许，能参与更多项目当然就更好了）。一个项目就是要编写一个完整的有用的程序。理想情况，一个项目由一个多人小组（比如三个人）共同完成，最好在学习第三部分的同时花大概一个月时间来完成整个项目。大多数人会发现做项目非常有趣，并在这个过程中学会如何把很多事情组织在一起。

一些人喜欢在读完一章之前就把书扔到一边，开始尝试做一些实例程序；另一些人则喜欢把一章读完后，再开始编码。为了帮助前一种读者，我们在正文的段落之间放置了一些有“试一试”标识的文字，给出了对于编程实践的一些简单建议。“试一试”本质上来说就是一个简单练习，而且只着眼于前面刚刚介绍的主题。如果你略去一个“试一试”而没有去尝试它（也许因为你的手边没有计算机，或者你过于沉浸在正文的内容中），那么最好在做这一章的简单练习时做一下这个题目。“试一试”要是该章简单练习的补充，要么干脆就是其中一部分。

在每章末尾你都会看到一些思考题，我们设置这些思考题是想为你指出这一章中的重点内容。一种学习思考题的方法是把它们作为习题的补充：习题关注程序设计的实践层面，而思考题则试图帮你强化思想和概念。因此，思考题有点像面试题。

每章最后都有“术语”一节，给出本章中提出的程序设计或 C++ 方面的主要词汇表。如果你希望理解别人关于程序设计的陈述，或者想明确表达出你自己的思想，就应该首先弄清术语表中每个术语的含义。

重复是学习的有效手段，我们希望每个重要的知识点都在书中至少出现两次，并通过习题再次强调。

0.1.3 进阶学习

当你完成本书的学习时，是否能成为一名程序设计和 C++ 方面的专家呢？答案当然是否定的！如果做得好的话，程序设计会是一门建立在多种专业技能上的精妙的、深刻的、需要高度技巧的艺术。你不能奢望花四个月时间就成为一名程序设计专家，就像你不能奢望花四个月、半年或一年时间就成为一名生物学专家、数学家、自然语言（如中文、英文或丹麦文）方面的专家或者小提琴演奏家一样。如果你认真地学完了这本书，你可以期待，也应该期待的是：你已经在程序设计领域有了一个很好的开始，已经可以写相对简单的、有用的程序，能读更复杂的程序，而且已经为进一步的学习打下了良好的理论和实践基础。

学习完这门入门课程后，最好的进一步学习的方法是开发一个真正能被别人使用的程序。在完成这个项目这之后，或者同时（同时可能更好），学习一本专业水平的教材（如 Stroustrup 的《The C++ Programming Language》），学习一本与你做的项目相关的更专业的书（例如，你如果在做 GUI

相关项目的话,可选择关于 Qt 的书;如果在做分布式程序的话,可选择关于 ACE 的书),或者学习一本专注于 C++ 某个特定方面的书(如 Koenig 和 Moo 的《Accelerated C++》^①, Sutter 的《Exceptional C++》^②或 Gamma 等人的《Design Patterns》^③)。完整的参考书目,参见 0.6 节或本书最后的“参考书目”一节。

最后,你应该学习另一门程序设计语言。我们认为,如果只懂一门语言,你是不可能成为软件领域的专家的(即使你并不想做一名程序员)。

0.2 讲授和学习本书的方法

我们是如何帮助你学习的?又是如何安排学习进程的?我们的做法是,尽力为你提供编写高效的实用程序所需的最基本的概念、技术和工具,包括:

- 程序组织
- 调试和测试
- 类设计
- 计算
- 函数和算法设计
- 绘图方法(仅介绍二维图形)
- 图形用户界面(GUI)
- 文本处理
- 正则表达式匹配
- 文件和流输入/输出(I/O)
- 内存管理
- 科学/数值/工程计算
- 设计和编程思想
- C++ 标准库
- 软件开发策略
- C 语言程序设计技术

认真完成这些内容的学习,我们会学到如下程序设计技术:过程式程序设计(C 语言程序设计风格)、数据抽象、面向对象程序设计和泛型程序设计。本书的主题是程序设计,也就是表达代码意图所需的思想、技术和工具。C++ 语言是我们的主要工具,因此我们比较详细地描述了很多 C++ 语言的特性。但请记住, C++ 只是一种工具,而不是本书的主题。本书主题是“用 C++ 语言进行程序设计”而不是“C++ 和一点程序设计理论”。

我们介绍的每个主题都至少服务于两个目的:提出一种技术、概念或原理,介绍一种实用的语言特性或库特性。例如,我们用一个二维图形绘制系统的接口展示如何使用类和继承。这使我们节省了篇幅(也节省了你的时间),并且还强调了程序设计不只是简单地将代码拼装起来以尽快地得到一个结果。C++ 标准库是这种“双重作用”例子的主要来源,其中很多主题甚至具有三重作用。例如,我们会介绍标准库中的向量类 `vector`,用它来展示一些广泛使用的设计技术,并展示

① 本书英文影印版和中文版均已由机械工业出版社引进出版。——编辑注

② 本书英文影印版已由机械工业出版社引进出版。——编辑注

③ 本书中文版已由机械工业出版社引进出版,书名为《设计模式》。——编辑注

很多用来实现 vector 的程序设计技术。我们的一个目标是向你展示一些主要的标准库功能是如何实现的，以及它们如何与硬件相配合。我们坚持认为一个工匠必须了解他的工具，而不是仅仅把工具当做“有魔力的东西”。

对于程序员来说，总是会对某些主题比其他主题更感兴趣。但是，我们建议你不要预先判断你需要什么(你怎么知道你将来会需要什么呢)，至少每一章都要浏览一下。如果你学习本书是作为一门课程的一部分，你的老师会指导你如何选择学习内容。

我们的教学方法可以描述为“深度优先”，也可以描述为“具体优先”和“基于概念”。首先，我们快速地(好吧，是相对快速的，从第1章到第11章)将一些编写小的实用程序所需的技巧提供给你。在这期间，我们还简明扼要地提出很多工具和技术。我们着重介绍简单具体的代码实例，因为相对于抽象概念，人们能更快地领会具体实例，这是大多数人的学习方法。在最初阶段，你不期望理解每个小的细节。特别是你会发现，对刚刚还工作得好好的程序只是稍加改动，便会呈现出“神秘”的效果。尽管如此，你还是要尝试一下！还有，请完成我们提供的简单练习和习题。请记住，在学习初期你只是没有掌握足够的概念和技巧来准确判断什么是简单的，什么是复杂的；请等待一些惊奇的事情发生，并从中学习吧。

我们会快速通过这样一个初始阶段——我们想尽可能快地带你进入编写有趣程序的阶段。有些人可能会质疑，“我们的进展应该慢些、谨慎些，我们应该先学会走，再学跑！”但是你见过小孩儿学习走路吗？小孩儿确实是在学会平稳地慢慢走路之前就开始自己学着跑了。与之相似，你可以先勇猛向前，偶尔摔一跤，从中获得编程的感觉，然后再慢下来，获得必要的精确控制能力和准确的理解。你必须在学会走之前就开始跑！

你不要投入大量精力试图学习一些语言或技术细节的所有相关内容。例如，你可以熟记所有 C++ 的内置类型及其使用规则。你当然可以这么做，而且这么做会使你觉得自己很博学。但是，这不会使你成为一名程序员。如果你学习中略过一些细节，将来可能偶尔会因为缺少相关知识而被“灼伤”，但这是获取编写好程序所需的完整知识结构的最快途径。注意，我们的这种方法本质上就是小孩儿学习母语的方法，也是教授外语的最有效的方法。有时你不可避免地被难题困住，我们鼓励你向授课老师、朋友、同事、辅导员以及导师等寻求帮助。请放心，在前面这些章节中，所有内容本质上都不困难。但是，很多内容是你所不熟悉的，因此最初可能会感觉有点难。

随后，我们向你介绍一些入门技巧，来拓宽你的知识和技巧基础。你通过实例和习题来强化理解，我们为你提供一个程序设计的概念基础。

我们重点强调思想和原理。思想能指导你求解实际问题——可以帮助你知道在什么情况下问题求解方案是好的、合理的。你还应该理解这些思想背后的原理，从而理解为什么要接受这些思想，为什么遵循这些思想会对你和使用你的代码的用户有帮助。没有人会满意“因为事情就是这样的”这种解释。更为重要的是，如果真正理解了思想和原理，你就能将已有的知识推广到新情况，就能用新的方法将思想和工具结合来解决新的问题。知其所以然是学会程序设计技巧所必需的。相反，仅仅不求甚解地记住大量规则和语言特性有很大局限，是错误之源，也是在浪费时间。我们认为你的时间很珍贵，尽力不浪费它。

我们把很多 C++ 语言层面的技术细节放在了附录和手册中，你可以随时按需查找。我们假定你有能力查找到你需要的信息，你可以借助索引和目录来查找信息。不要忘了编译器和互联网也有在线帮助功能。但要记住，要对所有互联网资源保持足够高的怀疑，直至你有足够的理由相信它们。因为很多看起来很权威的网站实际上是由程序设计新手或者想要出售什么东西的人建

立的。而另外一些网站，其内容都是过时的。我们在支持网站 www.stroustrup.com/Programming 上列出了一些有用的网站链接和信息。

请不要过于急切地期盼“实际的”例子。我们理想的实例都是能直接说明一种语言特性、一个概念或者一种技术的简短的代码。很多现实世界中的实例比我们给出的实例要凌乱很多，而且所能展示的知识也不如我们的实例更多。数十万行规模的成功商业程序中所采用的技术，我们用几个 50 行规模的程序就能展示出来。最快的理解现实世界程序的途径是好好研究一些基础的小程序。

另一方面，我们不会用“cute 风格”来阐述我们的观点。我们假定你的目标是编写供他人使用的实用程序，因此书中给出的实例要么是来说明语言特性，要么是从实际应用中提取出来的。我们的叙述风格都是用专业人员对(将来的)专业人员的那种口气。

0.2.1 本书内容顺序的安排

讲授程序设计有很多方法。很明显，我们不赞同“我学习程序设计的方法就是最好的学习方法”这种流行的看法。为了方便学习，我们较早地提出一些仅仅几年前还是先进技术的内容。我们的设想是，本书内容的顺序完全由你学习程序设计过程中遇到的问题来决定，随着你对程序设计的理解和实际动手能力的提高，一个主题一个主题地平滑向前推进。本书的叙述顺序更像一部小说，而不是一部字典或者一种层次化的顺序。

一次性地学习所有程序设计原理、技术和语言功能是不可能的。因此，你需要选择其中一个子集作为起点。更一般地，一本教材或一门课程应该通过一系列的主题子集来引导学生。我们认为选择适当的主题并给出重点是我们的责任。我们不能简单地罗列出所有内容，必须做出取舍；在每个学习阶段，我们选择省略内容与选择保留内容至少同样重要。

作为对照，这里列出我们决定不采用的教学方法(仅仅是一个缩略列表)，可能对你有帮助：

- C 优先：用这种方法学习 C++ 完全是浪费学生的时间，学生能用来求解问题的语言功能、技术和库比所需的要少得多，这样的编程实践很糟糕。与 C 相比，C++ 能提供更强的类型检查，对新手来说更好的标准库，以及用于错误处理的异常机制。
- 自底向上：学生本该学习好的、有效的程序设计技巧，但这种方法分散了学生的注意力。学生在求解问题过程中所能依靠的编程语言和库方面的支持明显不足，这样的编程实践质量很低、毫无用处。
- 如果你介绍某些内容，就必须介绍它的全部：这实际上意味着自底向上方法(一头扎进涉及的每个主题，越陷越深)。这种方法硬塞给初学者很多他们并不感兴趣，而且可能很长时间内都用不上的技术细节，令他们厌烦。这样做毫无必要，因为一旦学会了编程，你完全可以自己到手册中查找技术细节。这是手册适合的用途，如果用来学习基本概念就太可怕了。
- 自顶向下：这种方法，对一个主题从基本原理到细节逐步介绍，倾向于把读者的注意力从程序设计的实践层面上转移开，迫使读者一直专注于上层概念，而没有任何机会实际体会这些概念的重要性。例如，如果你没有实际体验编写程序是那么容易出错，而修正一个错误是那么困难，你就无法体会到正确的软件开发原理。
- 抽象优先：这种方法专注于一般原理，保护学生不受讨厌的现实问题限制条件的困扰，这会导致学生轻视实际问题、语言、工具和硬件限制。通常，这种方法基于“教学用语言”——一种将来不可能实际应用，有意将学生与实际的硬件和系统问题隔绝开的语言。
- 软件工程理论优先：这种方法和抽象优先的方法具有与自顶向下方法一样的缺点：没有具体实例和实践体验，你无法体会到抽象理论的价值和正确的软件开发实践技巧。

- 面向对象先行：面向对象程序设计是组织代码和开发工作的最好方法，但并不是唯一有效的方法。特别是，以我们的体会，在类型系统和算法式编程方面打下良好的基础，是学习类和类层次设计的前提条件。本书确实一开始就使用了用户自定义类型（一些人称之为“对象”），但我们直到第6章才介绍如何设计一个类，而直到第12章才介绍类层次。
- “相信魔法”：这种方法只是向初学者展示强有力的工具和技术，但不介绍其下蕴含的技术和功能。这让学生只能去猜这些工具和技术为什么会有这样的表现，使用它们会付出多大代价，以及它们合理的应用范围，通常学生会猜错！这会导致学生过分刻板地遵循相似的工作模式，成为进一步学习的障碍。

自然，我们不会断言这些我们没有采用的方法毫无用处。实际上，在介绍一些特定的内容时，我们使用了其中一些方法，学生能体会到这些方法在一些特殊情况下的优点。但是，当学习程序设计是以实用为目标时，我们不把一些方法作为一般的教学方法，而是采用其他方法：主要是具体优先和深度优先方法，并对重点概念和技术加以强调。

0.2.2 程序设计和程序设计语言

我们首先介绍程序设计，把程序设计语言作为一种工具放在第二位。我们介绍的程序设计方法适用于任何通用的程序设计语言。我们的首要目的是帮助你学习一般概念、原理和技术，但是不能孤立地学习这些内容。例如，不同程序设计语言在语法细节、编程想法的表达以及工具等方面各不相同。但对于编写无错代码的很多基本技术，如编写逻辑简单的代码（参见第5章和第6章），构造不变式（参见9.4.3节），以及接口和实现细节分离（参见9.7节和14.1节~14.2节）等，不同程序设计语言则差别很小。

程序设计技术的学习必须借助于一门程序设计语言，设计、组织代码和调试等技巧是不可能从抽象理论中学到的。你必须用某种程序设计语言编写代码，从中获取实践经验。这意味着你必须学习一门程序设计语言的基本知识。这里说“基本知识”是因为，花几个星期就能掌握一门主流实用编程语言全部内容的日子已经一去不复返了。本书讲述的与C++语言相关的内容只是它的一个子集，即与编写高质量代码关系最紧密的那部分内容。而且，我们所介绍的C++特性都是你肯定会用到的，因为这些特性要么是出于逻辑完整性的要求，要么是C++社区中最常见的。

0.2.3 可移植性

编写运行于多种平台的C++程序是很常见的情况。一些重要的C++应用甚至运行于我们闻所未闻的平台上！我们认为可移植性和对多种平台架构和操作系统的利用是非常重要的特性。本质上，本书的每个例子不仅是ISO标准C++程序，而且是可移植的。除非特别指出，本书的代码都能运行于任何一种C++实现，并且确实已经在多种计算机平台和操作系统上测试通过了。

不同系统编译、连接和运行C++程序的细节各不相同，如果每当提及一个实现问题时，就介绍所有系统和所有编译器的细节，是非常单调乏味的。我们在附录C中给出了Windows平台Visual Studio和Microsoft C++入门的大部分基本知识。

如果你在使用任何一种流行的，但相对复杂的IDE（integrated development environment，集成开发环境）时遇到了困难，我们建议你尝试命令行工作方式，它极其简单。例如，下面给出的是用GNU C++编译器，在UNIX或Linux系统中编译、连接和执行一个包含两个源文件my_file1.cpp和my_file2.cpp的简单程序所需的全部命令：

```
g++ -o my_program my_file1.cpp my_file2.cpp
my_program
```

是的，这真的就是全部。

0.3 程序设计和计算机科学

程序设计就是计算机科学的全部吗？答案当然是否定的！我们提出这一问题的唯一原因就是确实曾有人将其混淆。本书会简单涉及计算机科学的一些主题，如算法和数据结构，但我们的目标还是讲授程序设计：设计和实现程序。这比广泛接受的计算机科学的概念更宽，但也更窄：

- 更宽，因为程序设计包含很多专业技巧，通常不能归类于任何一种科学。
- 更窄，因为就我们涉及的计算机科学的内容而言，我们没有系统地给出其基础。

本书的目标是作为一门计算机科学课程的一部分（如果成为一个计算机科学家是你的目标的话），作为软件构造和维护领域第一门基础课程（如果你希望成为一个程序员或者软件工程师的话），总之是更大的完整系统的一部分。

本书自始至终都依赖计算机科学，我们也强调基本原理，但我们是以前理论和经验为基础来讲授程序设计，是把它作为一种实践技能，而不是一门科学。

0.4 创造性和问题求解

本书的首要目标是帮助你学会用代码表达你的思想，而不是教你如何获得这些思想。沿着这样一个思路，我们给出很多实例，展示如何求解问题。每个实例通常先分析问题，随后对求解方案逐步求精。我们认为程序设计本身是问题求解的一种描述形式：只有完全理解了一个问题及其求解方案，你才能用程序来正确表达它；而只有通过构造和测试一个程序，你才能确定你对问题和求解方案的理解是完整的、正确的。因此，程序设计本质上是理解问题和求解方案工作的一部分。但是，我们的目标是通过实例来说明这一切，而不是通过“布道”或是对问题求解详细“处方”的描述。

0.5 反馈方法

我们认为不存在完美的教材，个人的需求总是差别很大的。但是，我们愿意尽力使本书和支持材料更接近完美。为此，我们需要大家的反馈，脱离读者是不可能写出好教材的。请大家给我们发送反馈报告，包括内容错误、排版错误、含混的文字、缺失的解释等。我们也欢迎有关更好的习题、更好的实例、增加内容、删除内容等建议。大家提出的建设性的意见会帮助将来的读者，我们会将勘误表张贴在支持网站上：www.stroustrup.com/Programming。

0.6 参考文献

本节列出了本章提及的参考文献，以及可能对你有用的其他一些文献。

Austern, Matthew H. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley, 1999. ISBN 0201309564.

Austern, Matthew H. (editor). "Technical Report on C++ Standard Library Extensions." ISO/IEC PDTR 19768.

Blanchette, Jasmin, and Mark Summerfield. *C++ GUI Programming with Qt 4*. Prentice Hall, 2006. ISBN 0131872493.

Gamma, Erich, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN 0201633612.

Goldthwaite, Lois (editor). "Technical Report on C++ Performance." ISO/IEC PDTR 18015.

- Koenig, Andrew (editor). *The C++ Standard*. ISO/IEC 14882:2002. Wiley, 2003. ISBN 0470846747.
- Koenig, Andrew, and Barbara Moo. *Accelerated C++: Practical Programming by Example*. Addison-Wesley, 2000. ISBN 020170353X.
- Langer, Angelika, and Klaus Kreft. *Standard C++ IOSTreams and Locales: Advanced Programmer's Guide and Reference*. Addison-Wesley, 2000. ISBN 0201183951.
- Meyers, Scott. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley, 2001. ISBN 0201749625.
- Meyers, Scott. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*. Addison-Wesley, 2005. ISBN 0321334876.
- Schmidt, Douglas C., and Stephen D. Huston. *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*. Addison-Wesley, 2002. ISBN 0201604647.
- Schmidt, Douglas C., and Stephen D. Huston. *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Addison-Wesley, 2003. ISBN 0201795256.
- Stroustrup, Bjarne. *The Design and Evolution of C++*. Addison-Wesley, 1994. ISBN 0201543303.
- Stroustrup, Bjarne. "Learning Standard C++ as a New Language." *C/C++ Users Journal*, May 1999.
- Stroustrup, Bjarne. *The C++ Programming Language (Special Edition)*. Addison-Wesley, 2000. ISBN 0201700735.
- Stroustrup, Bjarne. "C and C++: Siblings"; "C and C++: A Case for Compatibility"; and "C and C++: Case Studies in Compatibility." *C/C++ Users Journal*, July, Aug., Sept. 2002.
- Sutter, Herb. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley, 2000. ISBN 0201615622.

更全面的参考文献列表可以在本书最后“参考书目”一节找到。

0.7 作者简介

你也许有理由问：“是一些什么人想要教我程序设计？”那么，下面是作者简介。我（即 Bjarne Stroustrup）和 Lawrence “Pete” Petersen 合著了本书。我还设计并讲授了面向大学生初学者（一年级学生）的课程，这门课程是与本书同步发展起来的，以本书的初稿作为教材。

Bjarne Stroustrup

我是 C++ 语言的设计者和最初的实现者。在过去的大约 30 年间，我使用 C++ 和许多其他程序设计语言进行过各种各样的编程工作。我喜欢那些用在富有挑战性的应用（如机器人控制、绘图、游戏、文本分析以及网络应用）中的优美而又高效的代码。我教过能力和兴趣各异的人设计、编程和 C++ 语言。我是 ISO 标准组织 C++ 委员会的创建者，现在我是该委员会语言演化工作组的主席。



这是我第一本入门级的书。我编著的其他书籍，如《The C++ Programming Language》和《The Design and Evolution of C++》都是面向有经验的程序员的。

我生于丹麦奥尔胡斯一个蓝领（工人阶级）家庭，在家乡的大学获得了数学与计算机科学硕士学位。我的计算机科学博士学位是在英国剑桥大学获得的。我为 AT&T 工作了大约 25 年，最初在著名的贝尔实验室的计算机科学研究中心——UNIX、C、C++ 及其他很多东西的发明地，后来在 AT&T 研究实验室。

我现在是美国国家工程院的院士，ACM 院士和 IEEE 院士，贝尔实验室院士和 AT&T 院士。我获得了 2005 年度 Sigma Xi 科学研究社区科学成就 William Procter 奖，我是首位获得此奖的计算

机科学家。

至于工作之外的生活，我已婚并有两个孩子，一个是医学博士，另一个目前是博士生。我喜欢阅读(包括历史、科幻、犯罪及时事等各类书籍)，还喜欢各种音乐(包括古典音乐、摇滚、蓝调和乡村音乐)。和朋友一起享受美食是我生活中必不可少的一部分，我还喜欢访问世界各地有趣的地方和人。为了能够享受美食，我还坚持跑步。

关于我的更多的信息，请浏览我的网站：www.research.att.com/~bs 和 www.cs.tamu.edu/people/faculty/bs。特别地，你可以在那里找到我名字的正确发音。

Lawrence “Pete” Petersen

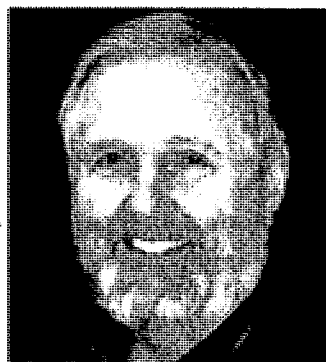
在2006年末，Pete如此介绍他自己：“我是一名教师。将近20年来，我一直在德州农工大学讲授程序设计。我已5次被学生选为优秀教师，并于1996年被工程学院的校友会选为杰出教师。我是Wakonse优秀教师计划的委员和教师发展研究院院士。

作为一名陆军军官的儿子，我的童年是在不断迁移中度过的。在华盛顿大学获得哲学学位后，我作为野战炮兵官员和操作测试研究分析员在军队服役了22年。1971年至1973年期间，我在俄克拉荷马希尔堡讲授野战炮兵军官的高级课程。1979年，我帮助创建了测试军官的训练课程，并在1978~1981年及1985~1989年期间在跨越美国的九个不同地方以首席教官的身份讲授这门课程。

1991年我组建了一个小型的软件公司，生产供大学院系使用的管理软件，直至1999年。我的兴趣在于讲授、设计和实现供人使用的实用软件。我在佐治亚理工大学获得了工业管理学硕士学位，在德州农工大学获得了教育管理硕士学位。我还从NTS获得了微型计算机硕士学位。我的信息和运营管理博士学位是在德州农工大学获得的。

我的妻子芭芭拉和我都生于德州的布莱恩。我们喜欢旅行、园艺和招待朋友；我们花尽可能多的时间陪我们的儿子们和他们的家人，特别是我们的几个孙子孙女，安吉丽娜、卡洛斯、苔丝、埃弗里、尼古拉斯和乔丹。”

令人悲伤的是，Pete于2007年死于肺癌。如果没有他，这门课程绝对不会取得成功。



附言

很多章都提供了一个简短的“附言”，试图给出本章所介绍内容的全景描述。这样做是因为我们意识到，知识可能是(而且通常就是)令人畏缩的，只有当完成了习题、学习了进一步的章节(应用了本章中提出的思想)并进行了复习之后才能完全理解。不要恐慌，放轻松，这是很自然的，也是可以预料的。你不可能一天之内就成为专家，但可以通过学习本书逐步成为一名相当胜任的程序员。学习过程中，你会遇到很多知识、实例和技术，很多程序员已经从中发现了令人激动的和有趣的东西。

第1章 计算机、人与程序设计

“只有昆虫才专业化。”

——R. A. Heinlein

在本章中，我们讲一些可以使程序设计变得重要和有趣的事情。我们也讲一些基本的思想与理想。我们希望揭穿几个有关程序设计与程序员的神话。本章是现在可以跳过的内容，当你困扰于一些编程问题和怀疑是否值得阅读时，可以返回阅读本章。

1.1 介绍

正如大多数的学习一样，学习程序设计就像鸡和蛋的问题。我们希望开始学习一件事，但是我们也希望了解为什么学习它。我们想学习一种实用技能，但是我们也希望确定它不只是短暂的流行。我们希望知道自己将不会浪费时间，但是我们也希望不被炒作和道德说教所打扰。现在，我们仅抱着有兴趣的态度来阅读本章，并在为什么技术细节会符合课堂外的情况，而感到要更新记忆时返回来阅读。

本章是对如何发现编程的兴趣与重要性的个人陈述。它解释了是什么激励我们数十年后还在这个领域中不断前进。通过阅读本章会得到可能的最终目标和程序员可能是哪种人的想法。针对初学者的技术书籍毫无疑问会包含很多基础的内容。在本章中，我们将眼睛从技术细节上移开，并且考虑一个更大的画面：为什么程序设计是一个有价值的活动？程序设计在我们的文明中扮演怎样的角色？程序员所做的贡献哪里值得骄傲？程序设计如何融入软件开发、部署和维护这一更大的领域？当人们谈论“计算机科学”、“软件工程”、“信息技术”等时，程序设计如何融入其中？一个程序员需要做什么？一个好的程序员需要具备哪些技能？

对于一个学生来说，理解一种思想、一项技术或一章的最紧迫的原因，可能是以好的成绩通过一次考试，但是在这里会有更多比学习重要的东西！对于那些在软件公司工作的人来说，理解一种思想、一项技术或一章的最紧迫的原因，可能是找到一些对目前的项目有帮助的东西，并且不会使控制你的薪水、升职和解雇的老板感到厌烦，但是在这里会有更多比学习重要的东西！当我们感到自己的工作会在细小的方面改善人们生活的世界，我们会努力将工作做到最好。对于那些在几年内完成的任务（在专业和职业发展中的“事情”），理想和更抽象的思想是决定性的。

我们的文明运行在软件之上。改进软件和发现软件的新用途是个人改进生活的两种方法。程序设计在这里扮演着一个基本的角色。

1.2 软件

好的软件是看不见的。你不能看到、感觉、称量或敲击它。软件是运行在计算机上的程序的集合。有时候我们可以看到一台计算机。我们通常仅仅看到由一些东西构成的计算机，例如一部电话机、一台照相机、一台面包机、一辆汽车或一台风力发电机。我们可以看到软件如何工作。如果软件没有按预想的方式工作，我们会感到厌烦或受到伤害。如果软件按预想的方式工作而不符合我们的需要，我们也会感到厌烦或受到伤害。

世界上有多少台计算机？我们并不知道，至少有数十亿台。世界上的计算机数量有可能超过人的数量。2004年，据ITU（国际电信联盟，一个联合国机构）的统计，共有7.72亿台个人计算机（PC），以及更多的不属于PC的计算机。

你每天会使用多少台计算机（直接或间接）？在一辆汽车中有超过30台计算机，在移动电话中有2台计算机，在MP3播放器中有1台计算机，在照相机中有1台计算机。我有自己的笔记本电脑与台式电脑。在夏天保持温度与湿度的空调也是1台简单的计算机。控制计算机科学系的电梯也是1台计算机。如果你使用的是现代的电视机，那么其中至少会有1台计算机。如果你进行一次网上冲浪，将会通过通信系统接触几十、也可能几百台服务器，通信系统中又包含数千台计算机（电话交换机、路由器等）。

我并不是在驾驶一辆后座上带着30台笔记本电脑的汽车！重点是这些计算机看起来不像通常的计算机（带有一个屏幕、一个键盘和一个鼠标等）；它们作为很小的一个“零件”嵌入我们使用的设备中。正因为如此，汽车中没有哪个东西看起来像计算机，即使是用于显示地图和行驶方向的屏幕（这种小工具在汽车中很流行）。但是，在汽车引擎中会包含几台计算机，用于完成燃油喷射控制与温度监控工作。汽车的助力转向系统包含至少1台计算机，广播与安全系统包含多台计算机，我们甚至怀疑车窗的开启/关闭都由计算机来控制。新型号的汽车甚至有用持续检测轮胎气压的计算机。

你在日常生活中所做的事情需要依赖于多少台计算机？你需要吃饭；如果你生活在一个现代化的城市中，为了将食物提供给你，需要计划、运输和存储。对这一分配网络的管理当然是计算机化的，它们之间通过通信系统连接起来。现代化农业也是高度计算机化的，你可以在牛舍附近发现用于监控牛群（年龄、健康、产奶量等）的计算机，农业设备也越来越计算机化。如果某些事情出错，你可以在报纸上阅读到它。当然，报纸上的文章通过计算机来书写，通过计算机来进行页面设置，以及通过计算机设备来印刷（如果你仍阅读纸质的报纸），通常需要以电子形式传输到印刷厂。书籍的生产采用的是同样的方式。如果你需要上下班，计算机通过监控交通流量以避免交通堵塞的。你喜欢乘坐火车？火车也是计算机化的。有些操作甚至不需要司机来完成，火车的子系统（广播、刹车和监票）包括很多计算机。今天的娱乐业（音乐、电影、电视、舞台表演）是大量使用计算机的用户。即使非卡通的电影也在大量使用（计算机）动画，音乐和摄影也趋向于数字化存储和传输（使用计算机）。如果你生病，医生为你做的检查要使用计算机，病历通常是计算机化的，大多数你遇到的用于治疗医学仪器也包含计算机。除非你碰巧住在树林中的草屋中，并且不使用任何电动工具（包括电灯），否则你都会使用能源。石油被发现、提炼、加工和传输的过程，从钻头深入地下到本地的汽油（天然气）加油站，整个过程中的每个步骤都要使用计算机。如果你使用信用卡来购买汽油，你也会访问一组计算机。对于煤炭、天然气、太阳能和风力发电，它们都会经过同样的过程。

迄今为止的例子都是“可操作的”，它们都直接包含在你所做的事情中。你未参与其中的事情是设计中的重要和有趣的部分。你穿着的衣服、你交谈用的电话和你调制自己喜欢的饮料用的咖啡机，这些都是通过计算机来设计与生产的。优质的现代摄影镜头、造型精美的日常工具和器具，这些几乎都要归功于基于计算机的设计与生产方式。那些设计我们周围环境的工匠、设计师、艺术家和工程师，他们从很多以前被认为是基本工作的物理限制中解脱出来。如果你生病，那些用来治愈你的药品也是使用计算机设计的。

最后，科学研究本身严重依赖于计算机。对于用于探索遥远的恒星秘密的望远镜，它们离开

计算机是无法设计、制造和操作的，它们产生的大量数据离开计算机是无法处理的。个别生物学领域的研究人员没有被严重计算机化(不包括照相机、数字录音机、电话等的使用)，但是回到实验室中，数据要使用计算机模型来储存、分析和检查，并且要和其他科研人员通信。现代化学和生物学(包括医学)大量使用计算机，在几年前就达到人们做梦也想不到的程度，并且至今对大多数人仍是难以想象的。人类基因测序是通过计算机完成的。让我们描述得更准确一些，人类基因测序是人使用计算机完成的。在所有这些例子中，我们可以看到计算机可以帮助我们完成某些事，在没有计算机的情况下需要花费更多时间。

每台计算机都需要运行软件。如果没有软件，计算机就是由硅、金属和塑料组成的昂贵的大块头，其与门吸、船锚和空间加热器没有多大区别。软件中的每行代码都是由不同的人编写的。如果软件在运行中有错误，实际执行的每行在合理的最低限度内。它们都正常运行是很惊人的事。我们谈论的是用几百种编程语言编写的几十亿行程序代码(或程序文本)。使它们都正常运行需要付出惊人的代价和令人难以想象的技巧。我们希望对所依赖的每种服务和工具进行更多的改变。思考一下你所依赖的某种服务和工具，你希望看到它们有怎样的改进？如果没有的话，我们希望服务和工具更小(或更大)、更快、更可靠、更有特点、更容易使用、更高性能、更好看和更便宜。这些我们想做的改进的相似点是都需要编程。

1.3 人

计算机是人制造的，供人使用的。计算机是一种非常通用性的工具，它可以被用于很多你无法想象的任务。计算机运行一个程序，以使它对一些人有用。换句话说，计算机只是一个硬件，直到一些人(程序员)编写代码使它能做某些事情。我们经常会忘记软件，甚至经常会忘记程序员。

好莱坞和类似的“流行文化”等谣言的来源已经给程序员带来很大的负面影响。例如，我们总是看到孤独的、肥胖的、丑陋的、不懂社交技巧的讨厌鬼，并且总是痴迷于视频游戏和闯入其他人的计算机。他(几乎总是男人)可能是想毁灭世界，也可能是想拯救世界。很明显，这种漫画人物的“温和版本”在现实生活中确实存在，但是依我们的经验他们中的软件开发人员，并不比律师、警官、汽车销售员、记者、艺术家或政治家更多。

我们思考一下计算机在现实生活中的应用。它们是在一个黑屋子中独立工作吗？当然不是，一个成功的软件、计算机设备或系统的创建，需要包括几十、几百或几千人扮演一系列扑朔迷离的角色，例如程序员、(程序)设计者、测试人员、美工人员、开发小组管理者、实验心理学家、用户界面设计者、分析人员、系统管理员、客户关系人员、音效工程师、项目经理、质量工程师、统计人员、硬件接口工程师、需求分析工程师、安全主管、数学家、销售支持人员、答疑人员、网络设计人员、方法论学家、软件工具管理员、软件库管理员等。这些角色的范围很广，随着组织之间的变换使人更加迷惑。一个组织中的“工程师”可能是另一个组织中的“程序员”，也可能是另一个组织中的“开发人员”、“技术组成员”或“结构设计师”。这里有多组织，不同成员可以在其中找到自己的头衔。并不是所有角色都与编程直接相关。但是，我们每个人都曾看到人们扮演每个角色的例子，他们将读写代码作为自己工作的重要部分。另外，一个程序员(扮演这些角色中的一个或多个)在短时期内会和不同应用领域的人打交道，例如生物学家、发动机设计师、律师、汽车销售员、医学研究员、历史学家、地理学家、宇航员、飞机工程师、木材库经理、火箭科学家、保龄球馆建设者、记者和漫画家(这是从个人经验中得到的)。有些人可能有时是一个程序

员，而在职业生涯的另一个阶段扮演非程序员的角色。

程序员是孤立的这一神话本身只是一个神话而已。那些喜欢工作在自己选择的工作领域的人，经常痛苦地抱怨被“打扰”或开会的次数。由于现代软件开发是一种团队行为，因此那些喜欢和别人交互的人会感到更轻松。这意味着社会和沟通能力是必不可少的，并且其价值远远大于陈规旧习。在一份对程序员很有用的技能的简短列表中(你是在从现实的角度定义程序员)，你会发现与不同背景的人进行沟通的能力最重要，这种沟通可能是非正式的会议、书面形式和正式介绍。我们相信，除非你完成过一个或两个团队项目，否则你不会知道什么是编程以及是否喜欢它。我们喜欢编程的理由是我们遇到的都是很好的、有趣的人，并且将访问风格各异的地方作为我们职业生涯的一部分。

所有这些是指那些有各种各样的技能、兴趣和工作习惯的人，对开发一个好的软件来说是必不可少的。我们的生活质量(有时甚至是生活自身)依赖于那些人。没有人可以扮演我们这里提到的所有角色，也没有明智的人希望扮演每个角色。重点是你有比自己所能想到的更大的选择空间，而不是不得不做出某个特定的选择。作为个人，你将“流向”那些符合你的技能、才智和兴趣的工作领域。

我们谈论的是“程序员”与“编程”，但是编程很明显只是整个画面的一个部分。那些设计船只或移动电话的人不会将自己做程序员。编程是软件开发中的一个重要部分，但并不是全部。相似地，对于大多数产品来说，软件开发是产品开发中的一个重要部分，但并不是全部。

我们并不假设你(我们的读者)希望成为一个专业程序员，将剩余的工作生涯用于编写代码。即使是那些优秀的程序员，他们也不会将大部分时间用在编写代码上。理解问题需要占用更多的时间，并且通常更消耗智力。智力挑战是很多程序员认为编程有趣的出发点。很多优秀程序员通常不是计算机科学专业的。例如，如果你进行基因研究方面的软件开发，理解分子生物学将会对你有更大的帮助。如果你进行中世纪文学分析方面的程序设计，阅读一些这类文学作品和掌握一门或多门相关语言将会对你有更大的帮助。特别是对于抱有“只关心计算机和编程”态度的人，他们将会难以与那些非程序员的同事交流。这样的人不仅会错过人与人交流(即生活)中最棒的那部分，而且也不会成为成功的软件开发人员。

于是，我们如何假设呢？编程是一种挑战智力的技能，需要经过很多重要与有趣的训练。另外，编程是我们这个世界的重要组成部分，不了解基本的编程知识就像不了解基本的物理、历史、生物或文学知识一样。那些对编程完全无知的人相信它是魔术，让这样的人承担很多技术工作是危险的。另外，编程可以带来乐趣。

但是，我们如何假设编程的用途？你也许将编程作为未来学习和工作的重要工具，而不是成为一个专业的程序员。你也许将与其他人进行专业 and 个人的交流，这些人可能是设计师、作家、经理或科学家，具有编程方面的基础知识将会有一定优势。你也许将专业水平的编程作为你学习和工作的一部分。即使你成为一个专业的程序员，也不意味着你除了编程之外不做任何事。

你可能成为一名计算机或计算机科学方面的工程师，但是这样也并不是“编程占据所有时间”。编程是用代码表达你的思想的方式，也是一种协助求解问题的方式。除非你有值得表达的思想和值得解决的问题，否则编程没有用处(纯粹是浪费时间)。

这是一本关于编程的书，我们曾经承诺本书可以帮助你学习如何编程，那么为什么我们要强调非编程的内容与编程的有限作用呢？一个优秀的程序员会理解代码和编程技术在一个项目中的作用。一个优秀的程序员(在多数情况下)是一个优秀的团队成员，并且会努力理解代码和其产

品如何很好地支持整个项目。例如,想象我在为一个新的 MP3 播放器进行编程,则我关心的只是代码优美程度和提供的简洁功能的数量。我可能一直在大型的、功能强大的计算机上运行这些代码。我可能对声音编码理论不屑一顾,因为它是“与编程无关的”。我将会待在自己的实验室里,而不是走出去与潜在的用户交流。这样,用户毫无疑问将对音乐有不好的体验,并且不欣赏图形用户界面(GUI)编程的最新发展。这样做的后果是可能对项目带来一场灾难。更大的计算机意味着更昂贵的 MP3 播放器和更短的电池寿命。编码是数字化音乐控制的重要部分,忽视编码技术的发展会导致每首歌曲增加所需的存储空间(不同编码获得相同品质的输出时的存储空间大小差异超过 100%)。无视用户的喜好(在你看来是奇怪的和过时的),通常会导致用户选择其他产品。编写一个好的程序的重要部分是理解用户的需求,并且在执行(即编码)时实现这些需求。为了完成上述对一个坏程序员的描绘,我倾向于将其描述成对细节的狂热和对简单测试的代码正确性的过分自信。我们鼓励你成为一个好的程序员,只有具有广阔的视野才能生产出好的软件。这既是社会价值也是个人自我实现之所在。

1.4 计算机科学

即使是在最广泛的定义中,也最好将编程看做某些更大事物的一部分。我们可以将编程看做计算机科学、计算机工程、软件工程、信息技术或其他软件相关的学科。我们将编程看做科学和工程中的计算机和信息领域的实现技术,同样也是物理学、生物学、医学、历史学、文学和其他学术或研究领域的实现技术。

请思考计算机科学。在 1995 年,美国政府的“蓝皮书”对它的定义如下:“对计算系统和计算的系统研究。这个学科造就的知识体系包含理解计算系统和方法的理论,设计方法学、算法和工具,测试概念的方法,分析和验证的方法,以及知识的表示和实现。”正如我们所预料的那样,维基百科条目所给出的概念不太正式:“计算机科学或计算科学是对信息和计算的理论基础的研究,以及它们在计算机系统中的应用。计算机科学包含很多子领域,有些强调特定结果的计算(例如计算机图形学),另一些关于计算问题的性能(例如计算复杂度理论)。有些集中在实现计算的挑战上。例如,编程语言理论研究描述计算的方法,而计算机编程使用特定的编程语言来解决特定的计算问题。”

编程是一种工具。它是一种针对基础和实践问题的基本工具,使这些问题可以通过实验来测试、改进和应用。编程是思想和理论的实际交汇。这是计算机科学可以成为一种实践训练而不是纯理论,并且影响世界的原因所在。在这方面,和很多其他事情一样,编程必不可少的是训练和实践的良好结合。它不应退化为这样一种活动:只是编写一些代码,用旧的方式满足眼前的简单需求。

1.5 计算机已无处不在

没有人知道计算机或软件相关的所有事。本节内容只是给出一些例子。你也许会看到自己想看的東西。你至少可以理解计算机的应用范围和编程所涵盖的范围远远超出任何个人可以完全掌握的程度。

大多数人认为计算机只是一个带有显示器和键盘的灰色盒子。这种计算机往往被放置在桌子下面,用于玩游戏、收发消息和邮件、播放音乐。另一些计算机称为笔记本电脑,它们被无聊的商人们在飞机上使用,查看报表、玩游戏和观看视频。这种讽刺性的描述只是冰山一角。大多

数的计算机工作在我们看不到的地方，并且作为保持社会运转的系统的一部分。它们中的一些可能占据整个房间，另一些可能比一枚小的硬币还小。这些有趣的计算机不是通过键盘、鼠标或类似的设备直接与人进行交互。

1.5.1 有屏幕和没有屏幕

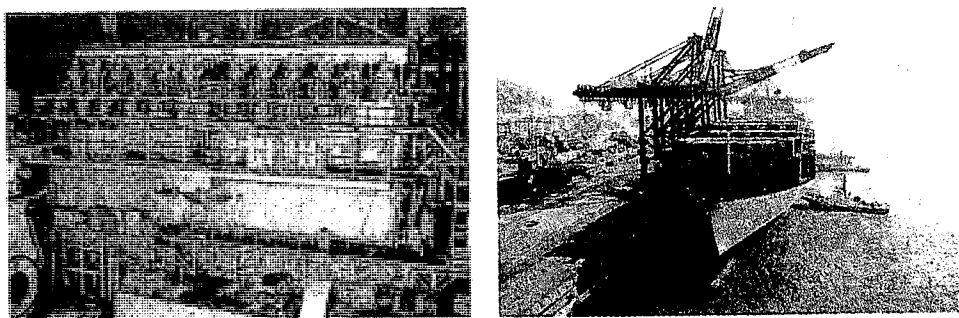
计算机是一个相当大的、带有屏幕和键盘的方盒子的想法很普遍，并且通常是难以动摇的。但是，我们考虑一下这两种计算机：



这两种用于计时的工具本质上也是计算机。实际上，我们猜测它们基本上是带有不同 I/O (输入/输出) 系统的相同模式计算机。左边那个驱动一个小的屏幕 (与普通计算机的屏幕类似，但是更小)，第二个驱动小的电子发动机来控制传统的表针和用于表示日、月的数字表盘。这些输入系统都有 4 个按钮 (右边那个更容易看清楚) 和 1 个无线电接收器，它被用于与非常精确的“原子”时钟保持同步。控制这两个计算机的大多数程序都是相同的。

1.5.2 船舶

这两张图片显示的是一台大型的船用柴油机和它可能驱动的巨大的船舶：



我们考虑一下计算机和软件在这里扮演的重要角色：

- 设计：当然，船舶和引擎都是使用计算机来设计的。这个过程中用到计算机的地方非常多，主要包括结构和工程制图、一般的计算、空间和部件的可视化以及性能的模拟。
- 建造：现代化的造船厂是高度计算机化的。船舶组装是通过计算机来严格规划的，工作是通过计算机来指导的。焊接是由机器人来完成的。特别是双壳油船，没有小的焊接机器人在壳体之间焊接是无法完成的。那里没有可以容纳一个人的空间。为船舶切割钢板是世界上最早的 CAD/CAM (计算机辅助设计和计算机辅助制造) 应用之一。
- 引擎：引擎采用电子燃料喷射技术，和由数十台计算机控制。对于一台 10 万马力的引擎 (就像照片中的那台)，这是一个非同一般的任务。例如，引擎管理计算机要持续调节燃

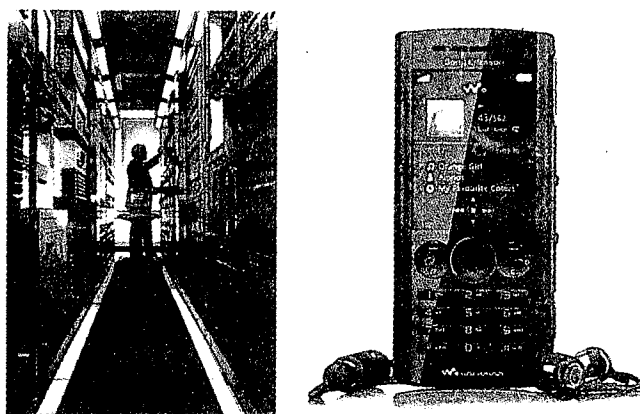
料注入,以便将引擎不协调可能导致的污染降到最小。很多与引擎相连接的泵(船舶的其他部分)本身也是计算机化的。

- **管理:** 船舶需要航行到特定的港口去装卸货物。船队的日程安排是一个持续的过程(当然是计算机化的),其目标是可以根据气象、供应和需求、港口的空间和吞吐量来调整航线。甚至有网站可以用来查询大型商船在某个时刻的位置。照片中的船舶碰巧是一艘集装箱船(这个世界上最大的船舶,397 米长和 56 米宽)。其他类型的大型现代化船舶都以相似的方式进行管理。
- **监控:** 一艘远洋船舶在很大程度上是自治的,它的全体船员可以在到达下一个港口前处理大多数可能产生的紧急事件。但是,它们仍是一个全球网络中的一部分。船员可以访问相当精确的气象信息(通过计算机化的人造卫星)。它们拥有 GPS(全球定位系统)和计算机控制、计算机增强的雷达。如果船员需要休息,大多数系统(包括引擎、雷达等)可以在航线控制室中进行监控(通过卫星)。如果发现任何异常或通信连接中断,船员会收到通知。

让我们考虑一下,如果在这段简短的介绍中明确提到或暗示的数百台计算机之一出现故障将意味着什么。在第 25 章(“嵌入式系统编程”)中,将会对这种情况做出稍微详细的解释。为一艘现代化船舶编写代码是一件讲究技巧且有趣的事。它也是很有用的。运输成本实际上是很低廉的。你在购买那些不在本地生产的東西时会赞成这一点。海洋运输总是比陆地运输更便宜,其主要原因在于计算机和信息的大量使用。

1.5.3 电信

这两张图片显示的是一台电话交换机和一部电话(它碰巧还是一台照相机、一台 MP3 播放器、一台 FM 收音机和一个 Web 浏览器):



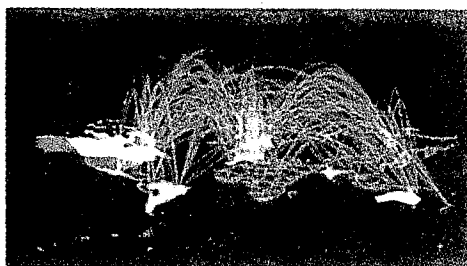
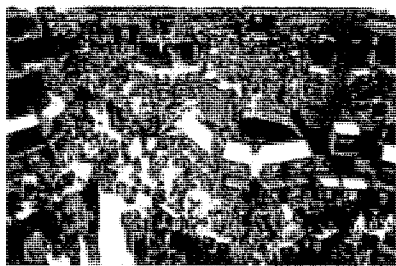
我们考虑一下计算机和软件在这里扮演的角色。你拿起一部电话拨号,你所拨叫的人响应,然后你们可以通话。你可能是在与一台应答器通话,可能通过电话中的照相机发送一张照片,或者发送一条文本消息(按“发送”使电话完成拨号)。很明显,电话是一台计算机。如果电话(像多数的移动电话)拥有一个屏幕,并且提供更多超过传统“老式电话服务”的功能(如 Web 浏览器),则这种情况将会特别明显。实际上,这类电话通常包含多台计算机:一台用于管理屏幕,一台用于与电话系统通话,可能还会包含更多计算机。

计算机用户最熟悉的可能是电话中的管理屏幕、进行 Web 浏览等:它为“所有常见的功能”提供一个图形化用户界面。大多数用户不知道、甚至感到意外的是与小的电话协作,完成通话工作

背后的庞大系统。我拨叫一个德克萨斯州的号码，而这时你正在纽约城度假，但是你的电话铃声在几秒钟内响起，并且我听到你伴着城市交通的嘈杂声说“你好！”。很多电话可以在地球上的两个位置之间通话，我们认为这是理所当然的。我的电话如何找到你的电话？声音如何被传输？声音如何被编码加入数据包？这些问题的答案可以填满比本书更厚的几本书，但是它会涉及分布在相关地理区域中的数百台计算机中的软件和硬件。如果你是不幸的，还会涉及几个通信卫星（它们也是计算机化的）。“不幸”的原因是我们不能完全补偿进入 2 万英里的太空的代价，光速（因此是你的声音传播的速度）是有限的（光纤电缆更好：更短、更快和传输更多数据）。这些工作多数是很好的，骨干通信系统的可靠性可以达到 99.9999%（例如，在 20 年中有 20 分钟断线，等于 $20/20 \times 365 \times 24 \times 60$ ）。我们遇到的麻烦通常出现在移动电话和最近的电话交换机之间的通信。

在这里，软件用于在电话之间建立连接，用于将语音编码为数据包通过有线或无线链路传输，用于路由这些消息，用于恢复各种故障，用于持续监控服务的质量和可靠性，当然也用于记账。跟踪系统中所有物理部分，也需要大量智能软件：谁和谁通话？哪部分进入一个新的系统？何时需要进行一些预防性维护？

这个世界的主干通信系统由很多半独立但互连的系统组成，它可能是最大和最复杂的人工产品。为了使事情更真实一些，记住，这不只是令人厌烦的老式电话带有一些新的铃声或哨音。各种基础设施已经合并。它们也是 Internet(Web)、金融和贸易系统、广播电台播放的电视节目运行的基础。因此，我们提供另一对图片来说明通信：



左图是位于纽约华尔街的美国证券交易所的“交易大厅”，右边的地图表示部分的 Internet 骨干网（一张完整的地图将会更加零乱）。

碰巧，我们也喜欢数字照片和用计算机绘制的地图来使知识可视化。

1.5.4 医疗

这两张图片显示的是一台 CAT（计算机轴向断层）扫描仪和一间计算机辅助手术室（也称为“机器人辅助手术”或“机器人手术”）：

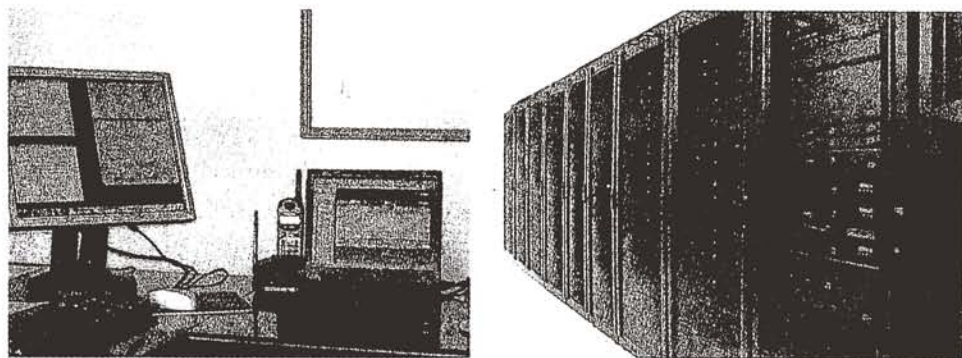


考虑一下计算机和软件在这里扮演的重要角色。扫描仪基本上就是计算机，它发出的脉冲由一台计算机来控制，它读取的内容对我们来说是杂乱无章的，除非将其通过复杂的算法转换成我们可以识别的人体相应部位的(三维)图像。为了进行计算机化的手术，必须分为几个步骤。各种成像技术用于使外科医生看清患者的身体内部，以便尽可能大和更亮地看清手术的部位。外科医生借助计算机可以使用人手无法握住的工具，或者不必割开身体就可到达某些部位。微创手术(腹腔镜手术)是一个最简单的例子，它减少了数百万人的痛苦和康复时间。计算机可以帮助稳定外科医生的“手”，以便完成正常情况下不可能完成的更细致的工作。最后，“机器人”系统可以远程操作，因此医生有可能远程(通过 Internet)医治病人。计算机和编程是难以置信的、复杂的和有趣的。用户界面、设备控制、成像技术中的每一项，都足以让数千名研究人员、工程师和程序员忙碌几十年。

我们听到很多医生关于哪种新的工具对他们的工作最有帮助的讨论：CAT 扫描仪？MRI 扫描仪？自动血液分析仪？高分辨率超声波仪？PDA？在经过讨论以后，令人惊讶的“胜利者”从这场“竞争”中出现：即时访问病历。了解患者的医疗史(早期疾病、早期用药、过敏史、遗传问题、一般健康状况、当前用药等)会简化问题的诊断和减小发生错误的机会。

1.5.5 信息领域

这两张图片显示的是一台普通 PC(也可能是两台)和服务器机群的一部分：



我们曾经将注意力集中在普通用途的“工具”上：你不能看到、感觉到或听到软件。我们无法给你提供一张程序的图片，因此我们给你看一下运行它的“工具”。但是，很多软件直接处理“信息”。因此，让我们来考虑一下运行“普通软件”的“普通计算机”的“普通用途”。

一个“服务器机群”是提供 Web 服务的多台计算机的集合。通过使用 Web 搜索引擎，我们找到由维基百科(一个 Web 目录)提供的下列知识。在 2004 年，据估计搜索引擎的服务器机群是以下规模：

- 719 个机架
- 63272 台机器
- 126544 个 CPU
- 253THz 的处理能力
- 126544GB 的内存
- 5062TB 的硬盘空间

一个 GB 是 1G 字节，大约是 1000000000 个字符。一个 TB 是 1T 字节，等于 1000 GB，大约是 1000000000000 个字符。最近，这个“机群”变得更加庞大。这是一个相当极端的例子，但是每个

大公司都在 Web 上运行程序,并通过它与用户或消费者进行交互。更多的例子包括 Amazon(销售图书和其他商品)、Amadeus(航空票务和汽车租赁)和 eBay(在线拍卖)。数以百万计的小公司、组织和个人也存在于 Web 上。它们中的大多数并不运行自己的软件,但是也有很多在运行并且相当复杂。

其他更传统、更大规模的计算机应用主要涉及:会计、订单处理、发薪水、记录保存、账单、库存管理、个人记录、学生记录、病历等,基本上每个组织都需要保存记录(商业和非商业、政府和个人)。这些记录是每个组织的支柱。通过计算机处理这些记录看起来很简单:这些信息(记录)中的大多数只需要存储和检索,只有非常少的部分需要处理。这方面的例子主要包括:

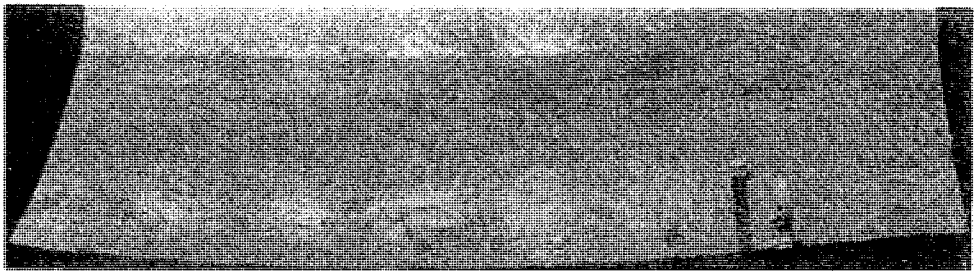
- 12:30 飞往芝加哥的航班是否仍然准时?
- Gilbert Sullivan 是否曾经患过麻疹?
- Juan Valdez 订购的咖啡是否已经启运?
- Jack Sprat 在 1996 年购买的是哪种餐椅?
- 2006 年 8 月从 212 区号拨出电话的数量是多少?
- 1 月售出的咖啡壶数量和总价是多少?

规模庞大的数据库使得这些系统非常复杂。这样,就对响应时间(对每个查询的响应通常不超过 2 秒钟)和准确性(至少在大多数情况下)的需求。如今,人们谈论 T 字节的数据(一个字节等于用于存储一个普通字符的内存大小)已经很常见了。这就是传统的“数据处理”,它正在和“Web”相融合,这是由于当前多数的数据库访问都通过 Web 接口。

这种计算机应用通常称为信息处理。它将重点集中在数据上,通常是大量的数据。这就导致了在数据的组织和传输上的挑战,以及在怎样以可以理解的形式来表示大量数据的大量有趣的工作:“用户接口”是处理数据中的重要方面。例如,对古典文学(Chaucer 的《Canterbury Tales》或 Cervantes 的《Don Quixote》)的分析工作,通过比较几十个版本以找出哪个才是作者的实际创作。我们需要以分析人员提供的多种标准来搜索文本,并且以有助于发现要点的方式来显示结果。思考一下文本分析和出版:当前,几乎所有的文章、书籍、小册子、报纸等都通过计算机生产。设计出能够很好地支持这一切的软件,对大多数人仍是一个缺乏真正好的解决方案的问题。

1.5.6 一种垂直的视角

有人曾经宣布古生物学家可以重构一个完整的恐龙,并且通过研究一块小的骨骼来描述它的生活方式和自然环境。这有可能是一种夸张的说法,但是从中可以体会到通过观察一个简单的产品来思考它的含义的思想。我们考虑一下这张显示火星风景的照片,它由 NASA 的火星探测器携带的照相机所拍摄:



如果你希望研究“火箭科学”,那么成为好的程序员是一种方式。各种空间计划需要大量软件设计人员,特别是根据载人或非载人空间计划需要的懂得物理、数学、电子工程、机械工程、医疗工程

等的人员。人类已成功发射两颗探测器围绕火星运行四年多(估计的设计寿命是3个月),这是人类文明最伟大的成就技术之一。

这张照片通过一条通信信道经过单向25分钟的传输延时传输到地球,这里需要很多巧妙的编程和高等数学应用,以便保证以最少的比特数、同时无差错地传输图片。在地球上,通过某些算法对这张照片进行渲染以恢复颜色和减小失真,这些问题都是由光学和电子传感器引起的。

火星探测器的控制程序当然也是程序,探测器每24小时会自动驱动一次,执行前一天从地球发送的指令。数据传输也是由程序来管理的。

探测器中的各种计算机使用的操作系统、传输和照片重构都是程序,在这点上和用来编写本章的各种应用程序相似。运行这些程序的计算机是通过CAD/CAM(计算机辅助设计和计算机辅助制造)程序来设计和生产的。这些计算机中的芯片是通过计算机化生产线用精密工具组装的,这些工具在它们的设计和制造中也使用计算机(或软件)。对这个长期组装过程的质量监控涉及大量计算。所有这些代码都由程序员用高级编程语言编写,并且通过编译器(本身就是一个程序)转换成机器代码。很多程序使用GUI与用户进行交互,以及使用输入/输出流进行数据交换。

最后,在图像处理(包括来自火星探测器的照片处理)、动画和照片编辑(在网络上有很多描述围绕“火星”的探测器照片)方面也需要大量编程工作。

1.5.7 与C++程序设计有何联系

这些“多样和复杂的”应用和软件系统与学习编程和使用C++有什么关系?这个关系很简单,很多程序员会参加这样的项目。这些事是好的程序设计可以帮助实现的。本章中用到的每个例子都涉及C++和本书中描述的几种技术。是的,在MP3播放器、船舶、风力发电机组、火星探测和人类基因工程中都会用到C++编程。如果想获得更多的使用C++的例子,你可以查看www.research.att.com/~bs/applications.html。

1.6 程序员的理想

我们希望从自己的程序中获得什么?我们通常(而不是从特定程序的特定功能)希望获得什么?我们希望保证正确性和作为其中一部分的可靠性。如果程序没有按照设想工作,没有按我们可以信赖的方式工作,小则是一个严重干扰、大则是一个危险。我们希望它得到良好的设计,这样它可以很好地满足实际需要;如果它所做的事情与我们无关,或者以某种我们厌烦的方式完成,则不能说程序是正确的。我们同样希望可以负担得起;我可能喜欢用Rolls-Royce汽车或行政专机作为日常交通工具,但是除非我是一名亿万富翁,否则开销会影响我的选择。

这些方面是软件(工具、系统)能得到外部的、非程序员的赞赏的所在。如果我们希望开发出成功的软件,那么这些内容必须成为程序员的理想,我们必须将它们永远记在心中,特别是在程序开发的早期阶段。此外,我们必须关注与代码本身相关的理想:我们的代码必须是可维护的,那些没有编写它的人可以理解它的结构和进行修改。一个成功的程序可以“生存”很长时间(通常有几十年)并经过反复修改。例如,它将会移植到新的硬件上,它将会增加新的功能,它将会修改以使用新的I/O设备(屏幕、视频、音频),它将会用新的自然语言进行交互等。只有失败的程序才永远不会被修改。为了保证可维护性,一个程序必须只与它的需求相关,它的代码必须直接体现要表达的思想。复杂性是简单性和可维护性的敌人,它对程来说可能是必需的(在那种情况下我们不得不处理它),但是也可能是由于没有用代码清晰地表达出思想而产生。我们必须通过良好的编码风格来尽量避免它——风格是很重要的!

这听起来不太难,但是做起来确实很难。为什么?编程基本上是简单的:就是告诉机器你打算做什么。但是,为什么编程中要面对很多挑战?计算机基本上是简单的,它只能做很少几种操作,例如两个数相加和基于两个数的比较来选择要执行的下一条指令。问题是我们并不希望计算机做简单的事情。我们希望“机器”帮助我们做那些难以完成的事,但是计算机是挑剔的、无情的和不会说话的东西。另外,这个世界要比我们所相信的更复杂,因此我们并不能理解自己需求的实际含义。我们只是希望一个程序能够“像这样做一些事情”,但是我们并不希望被技术细节所困扰。我们通常假设“基本常识”。不幸的是,人们认为很普通的基本常识,在计算机中通常完全不存在(通过某些精心设计的程序,可以在具体的、很好理解的情况下模拟它)。

这种思路导致的想法是“编程就是理解”:当你需要编写一个任务时,你需要理解它。相反,当你彻底理解一个任务,你可以编写程序去执行它。换句话说,我们可以将编程看做努力去彻底理解一个课题的一部分。程序是我们对一个课题的理解的精确表示。

当你在进行编程时,你会花费很多时间尝试理解你试图自动化的任务。

我们可以将描述开发程序的过程分为四个阶段:

- 分析:问题是什么?用户想要做什么?用户需要什么?用户可以负担什么?我们需要哪种可靠性?
- 设计:我们如何解决问题?系统的整体结构将是怎样的?系统包括哪些部分?这些部分之间如何通信?系统与用户之间如何通信?
- 编程:用代码表达问题(或设计)求解的方法,以满足所有约束(时间、空间、金钱、可靠性等)的方式编写代码。保证这些代码是正确的和可维护的。
- 测试:系统化地尝试各种况,保证系统在所要求的所有情况下都能正确工作。

编程和测试相加通常称为实现。很明显,将软件开发简单分为四个部分是一种简化。分别针对这四个主题编写的书都很厚,并且很多书仍在讨论它们之间的关系。需要说明的一件重要的事情是开发的这四个阶段并不是独立的,并且不一定严格按照顺序依次出现。我们通常从分析开始,但是通过测试的反馈有助于对编程的改进;编程工作带来的问题可能表明设计带来的问题;按设计进行工作可能发现在设计中至今仍被忽视的某些方面的问题。系统的实际使用通常会暴露分析中的一些弱点。

这里的关键概念是反馈。我们从经验中学习,根据学到的东西改变我们的行为。这是高效软件开发的根本。对于很多大的项目,我们在开始之前不可能理解有关问题的所有事情和解决方案。我们可以尝试自己的想法和从编程中得到反馈,但是在开发的早期阶段更容易(更快)从设计方案的书写、按设计思路编程和朋友的使用中得到反馈。我们知道的最好的设计工具是黑板。你要尽可能避免独立设计。在已将设计思路解释给其他人之前,不要开始进行编码工作。在接触键盘之前,与朋友、同事、潜在用户等讨论设计和编程技术。从简单尝试到阐明思路的过程中,你所学到的东西是令人惊讶的。最终,程序只不过是某些思路的表达(用代码)。

同样,当你实现一个程序时遇到问题,将目光从键盘上移开。考虑一下问题本身,而不是你的不完整的方案。与别人交流:解释你希望做什么和为什么它不工作。令人惊讶的是,你向有些人详细解释问题的过程中经常会找到解决方案。除非不得已,否则不要单独进行调试(找到程序错误)!

本书的重点是实现,特别是编程。我们不讲授“解决问题”,以及提供有关问题的足够例子和它们的解决方案。很多问题的解决是认识到一个已知的问题,以及使用一个已知的解决方案。只有当大多数子问题以这种方式解决后,你才可能专注于令人兴奋的和有创造性的“跳出固有模式

的思维”。因此，我们重点介绍如何用代码明确表达思路。

用代码直接表达思路是编程的基本的理想。这确实是很明显，但是至今我们还缺少好的例子。我们将会反复回到这里。当我们需要在自己的代码中使用一个整数时，我们将它保存在一个 `int` 类型中，它会提供基本的整数操作。当我们需要使用一个字符串时，我们将它保存在一个 `string` 类型中，它会提供基本的文本控制操作。在最基本的层次上，理想是当我们有一个思路、概念和实体时，即那些我们可以作为“事情”考虑的、可以写在黑板上的、可以加入讨论的、（非计算机科学）教科书中讨论的东西，我们需要这些东西在程序中作为一个命名实体（类型）存在，并且提供我们需要它们执行的操作。如果我们需要进行数学计算，则需要一个复数的 `complex` 类型和一个线性代数的 `Matrix` 类型。如果我们需要进行图形处理，则需要一个 `Shape` 类型、一个 `Circle` 类型、一个 `Color` 类型和一个 `Dialog_box` 类型。当我们处理来自比如说一个温度传感器的数据流时，我们需要一个 `istream` 类型（“i”表示输入）。很明显，每种类型将提供适当的操作，并且只提供适当的操作。这些只是本书中提到的几个例子。基于上述内容，我们提供用于构建自己的类型的工具和技术，以使用程序直接表达你希望体现的概念。

编程是实践和理论相结合的。如果你只重视实践，你将制造出不可扩展的、不可维护的程序。如果你只重视理论，你将制造出无法使用的（或无法负担的）玩具。

如果你想获得有关编程理想的不同类型的观点，以及少数在编程语言方面对软件做出重要贡献的人，请看第22章“理想和历史”。

思考题

思考题的目的是说明本章所解释的关键思想。可以把它们看做是练习的补充：练习关注的是编程的实践方面，而思考题尝试帮助你阐明思想和概念。在这方面，它们类似于好的面试题。

1. 什么是软件？
2. 软件为什么重要？
3. 软件重要在哪里？
4. 如果有些软件失败，那么导致错误的原因是什么？列举一些例子。
5. 软件在哪里扮演重要角色？列举一些例子。
6. 哪些工作与软件开发相关？列举一些例子。
7. 计算机科学和编程之间的区别是什么？
8. 在船舶的设计、建造和使用中，软件使用在哪些地方？
9. 什么是服务器机群？
10. 你在线提出哪种类型的查询？列举一些例子。
11. 软件在科学方面有哪些应用？列举一些例子。
12. 软件在医疗方面有哪些应用？列举一些例子。
13. 软件在娱乐方面有哪些应用？列举一些例子。
14. 我们期待中的好软件的一般特点有哪些？
15. 一个软件开发者看起来像什么？
16. 软件开发有哪些阶段？
17. 软件开发为什么困难？列举一些原因。
18. 软件的哪些用途为人类生活带来便利？
19. 软件的哪些用途为人类生活带来更多困难？

术语

这些术语是编程和 C++ 方面的基本词汇。如果你希望理解人们谈到的关于编程的主题和阐明自己的思

路,那么你应该知道每个术语的含义。

负担得起	客户	程序员	分析	设计	程序设计
黑板	反馈	软件	CAD/CAM	图形用户界面(GUI)	陈规旧习
沟通	理想	测试	正确性	实现	用户

习题

1. 选择一种你最常做的活动(例如上学、吃饭或看电视)。列举计算机直接或间接参与其中的方式。
2. 选择一种你最感兴趣或了解的职业。列举这些职业的人涉及计算机的活动。
3. 将你从习题2中得到的列表与选择不同职业的朋友交换,并且改进他或她的列表。当你完成这个练习,比较你们的结果。记住,这种开放式的练习没有完美的解决方案,永远有可能需要改进。
4. 根据你自己的经验,描述一种离开计算机不可能进行的活动。
5. 列举你已经使用过的程序(软件应用)。列举那些你与程序有明显交互的例子(例如在一台MP3播放器中选择一首新歌),而不是那些可能涉及计算机的例子(例如转动你的汽车方向盘)。
6. 列举十种完全不会涉及(即使是间接的)计算机的人类活动。这可能会比你想象得更难。
7. 列举五个当前没有使用计算机,但是你认为在将来会使用的任务。为你选择的每个任务写几句话加以说明。
8. 解释(至少100字,但不超过500字)为什么你想成为一名计算机程序员。另一方面,如果你认为自己不想成为一名程序员,请解释原因。在这两种情况下,提供深思熟虑、合乎逻辑的论据。
9. 解释(至少100字,但不超过500字)除了程序员之外,你希望在计算机工业中扮演的角色(“程序员”是否是你的首要选择)。
10. 你认为计算机将会发展到有意识、有思想、有能力与人类竞争的程度吗?写一段支持你的观点的话(至少100字)。
11. 列举最成功的程序员共有的特点。列举通常认为程序员应该具有的特点。
12. 列举五种在本章中提到的对计算机程序的应用,并选择一种你最感兴趣和将来最想参加的应用。并解释为什么选择这种应用(至少100字)。
13. 保存本页文字、本章、莎士比亚的所有著作各需要多大内存?假设1字节的内存可以保存一个字符,在这里只是尽量精确到20%。
14. 你的计算机拥有多大的内存?主存储器多大?磁盘多大?

附言

我们的文明运行在软件之上。软件是一种有趣的、对社会有益的、有利可图的工作,它是具有无与伦比的多样性和机会的领域。当你接触软件时,以有原则和严肃的方式工作:你要成为解决方案的一部分,而不是增加问题。

我们对贯穿技术文明的各种软件很敬畏。当然,软件并不是在所有应用中都表现良好,但那是另外一回事。在这里,我们想强调的是软件是多么普遍,以及我们在日常生活中多么依赖软件。它们都是由像我们这样的人来编写的。所有的科学家、数学家、工程师、程序员等,他们构建这里简略提到的软件的起点和你是一样的。

现在,让我们回到脚踏实地的业务上,学习那些编程需要的技术性的技能。如果你开始怀疑自己艰苦工作是否值得(大多数深思熟虑的人有时会怀疑它),你可以返回并重新阅读本章、前言和第0章。如果你开始怀疑自己是否能掌握这一切,请记住已经有数百万人成为称职的程序员、设计师、软件工程师等。你也可以做到。

第一部分 基本知识

第2章 Hello, World!

“程序设计要通过编写程序的实践来学习”

——Brian Kernighan

在本章中，我们提出最简单的 C++ 程序，它们实际上可以做任何事。编写这些程序的目的如下：

- 让你试用自己的编程环境。
- 给你一个最初的体验：如何让计算机为你做某些事。

因此，我们提出程序的概念，使用编译器将程序从人类可读的形式转换到机器指令，以及最终执行机器指令的思想。

2.1 程序

为了使计算机能够做某件事，你（或其他人）需要在繁琐的细节上明确告诉它怎么做。对“怎么做”的描述称为程序，编程是书写和测试这个程序的行为。

在某种意义上，我们以前都编写过程序。毕竟，我们曾描述过所要完成的任务，例如“如何开车去最近电影院”、“如何找到楼上的浴室”和“如何用微波炉热饭”。这种描述和程序之间的不同表现在精确度上：人类往往通过常识对不明确的指示加以补偿，但是计算机不会这样。例如，“沿走廊右转，上楼，它就在你的左边”可能是对如何找到楼上的浴室的很好描述。但是，当你看到这些简单的指令时，你会在其中找到草率的语法和不完整的指令。人类很容易做出补偿。例如，假设你坐在桌子旁询问浴室的方向。你不需要被告知离开桌子来到走廊、绕过（不是跨过或钻过）桌子、不要踩到猫等。你不需要被告知不要带走刀子和叉子，以及记住打开灯才能看到楼梯。你也不需要被告知进入浴室之前首先要开门。

与此相反，计算机确实是不很笨的。它们做的所有事都要准确、详细地描述。我们考虑“沿走廊右转，上楼，它就在你的左边”。走廊在哪里？什么是走廊？什么是“右转”？什么是楼梯？我如何上楼梯？（每次迈出一大步？两小步？沿扶手滑上楼梯？）什么是我的左边？它什么时候会在我的左边？为了向计算机精确描述这些“事情”，我们需要一种由特定语法精确定义的语言（英语对它来说有太多的松散结构）和针对我们要执行的多种行动明确定义的词汇。这种语言称为编程语言，C++ 是为各种编程任务而设计的编程语言。

如果你想知道有关计算机、程序和编程的更多哲学上的细节，请阅读第 1 章。在这里，让我

们来看一些代码，从一个很简单的程序和运行它的工具和技术开始学习。

2.2 经典的第一个程序

这是经典的第一个程序的一个版本。它在你的屏幕上输出“Hello, World!”:

```
// This program outputs the message "Hello, World!" to the monitor

#include "std_lib_facilities.h"

int main()    // C++ programs start by executing the function main
{
    cout << "Hello, World!\n";    // output "Hello, World!"
    return 0;
}
```

我们可以将这段文字看做是交给计算机执行的一组指令，就像我们交给一个厨师的一张菜谱，或我们用于使一个新玩具工作的一组指令集合。我们从这下面行开始讨论这个程序的每行如何工作。

```
cout << "Hello, World!\n";    // output "Hello, World!"
```

这是实际生成输出内容的一行。它打印字符串 Hello, World!, 并且紧跟换一个新行。也就是说，在打印出 Hello, World! 之后，光标将位于下一行的开始位置。光标是一个小的、闪烁的字符或行，它用来显示你可以输入下一个字符的位置。

在 C++ 中，字符串常量是由双引号(")来分隔。也就是说，“Hello, World! \n”是一个字符串。\\n 是一个用于指定新行的“特殊字符”。名称 cout 是一个标准的输出流。使用输出操作符 << “放入 cout”的字符将显示在屏幕上。名称 cout 的发音是“see-out”，它是“character output stream”的缩写。你会发现在编程时缩写很常见。很自然，在你第一次看到和要记住一个缩写时会觉得有点儿烦，但是当你开始重复使用一个缩写时，它们将会变得很自然，并且对保持程序文本的简短和可控制是必不可少的。

这行的结尾

```
// output "Hello, World!"
```

是一个注释。在一行中的//符号(/符号称为“斜杠”，这里是两个斜杠)之后的内容都是注释。注释会被编译器忽略，但对人们读懂代码很有帮助。在这里，我们使用注释告诉你这一行的开始部分实际在做什么。

注释用于描述这个程序打算做的事情，它通常提供的是对人们有用但是不能用代码直接来表达的信息。当你在下一星期或下一年回过头来阅读代码，并且已经忘记为什么以这种方式编写代码时，你最有可能通过代码中的注释得到帮助。因此，做好你的程序的文档工作。在 7.6.4 节，我们将讨论如何做好注释。

程序是为两个读者编写的。理所当然，我们编写代码是为了在计算机中执行。然而，我们阅读和修改代码也花费很长时间。于是，程序员是程序的另一个读者。因此，编写代码也是人与人之间沟通的一种方式。实际上，考虑将人类作为我们的代码的主要读者是有意义的：如果他们(我们)发现代码不是那么容易理解，那么代码永远不可能变得正确。总而言之，注释只是对人类读者有帮助的，计算机并不会看注释中的文本。

这个程序中的第一行是一个典型的注释，它简单地告诉人类读者程序打算做什么：

```
// This program outputs the message "Hello, World!" to the monitor
```

由于这段代码本身说明了程序做什么，而不是我们想让它做什么，因此这个注释是有用的。我们通常向人类(粗略地)解释一个程序将做什么，比我们用代码向计算机(详细)表达它要简单得多。这种

注释通常是我们编写的程序的第一部分。如果没有其他内容，它会提醒我们正在尝试做什么。

下一行

```
#include "std_lib_facilities.h"
```

是一个“`#include` 指令”。它指示计算机从名为 `std_lib_facilities.h` 的文件中提供(“包含”)功能。我们编写这个文件以简化使用所有 C++ 实现(“C++ 标准库”)中的功能。我们将随着学习的深入解释它的内容。它完全是普通的标准 C++ 程序，但是它包含我们不得不介绍的细节，在后续章中将会介绍它们。对于这个程序来说，`std_lib_facilities.h` 的重要性表现在我们可以使用标准 C++ 流 I/O 功能。在这里，我们只使用标准输出流 `cout` 和它的输出操作符 `<<`。使用 `#include` 包含的文件通常有后缀 `.h`，称为头或头文件。在头文件中包含术语的定义，例如在我们的程序中使用的 `cout`。

一台计算机如何知道从哪里开始执行一个程序？它会查找一个称为 `main` 的函数，并且在那里开始执行找到的指令。下面是“Hello, World!”程序的 `main` 函数：

```
int main() // C++ programs start by executing the function main
{
    cout << "Hello, World!\n"; // output "Hello, World!"
    return 0;
}
```

每个 C++ 程序必须有一个称为 `main` 的函数，以便告诉计算机从哪里开始执行。一个函数基本上是一个命名过的指令序列，计算机会按照它们的编写顺序来执行。一个函数包括 4 个组成部分：

- 返回值类型，在这里是 `int`(表示“整数”)，它用来指定返回结果的类型。如果有的话，这个函数会将值返回给要求它执行的程序。单词 `int` 在 C++ 中是保留字(一个关键字)，因此 `int` 不能用于作为其他任何东西的名字(见 A.3.1 节)。
- 名字，在这里是 `main`。
- 参数列表，封闭在一对括号中(见 8.2 节和 8.6 节)，在这里是 `()`，在这种情况下，参数列表是空的。
- 函数体，封闭在一对大括号中，在这里是 `{}` 中，列出了这个函数将要执行的动作(称为语句)。

下面是最简单的 C++ 程序

```
int main() {}
```

由于这个程序没有做任何事情，因此它并没有什么用处。我们的“Hello, World!”程序的 `main()`(“主函数”)体中有两条语句：

```
cout << "Hello, World!\n"; // output "Hello, World!"
return 0;
```

首先，它在屏幕上书写 `Hello, World!`，然后返回一个值 0(零)给它的调用者。由于 `main()` 是由“系统”来调用的，因此我们不会使用返回值。但是，在有些系统(特别是 UNIX/Linux)中，返回值可以用于检查程序是否成功。由 `main()` 返回的一个零(0)表示程序成功终止。

在 C++ 程序中用于指定一个行为并且不是一个 `#include` 指令(或其他预处理器指令，见 4.4 节和 A.17 节)的部分称为语句。

2.3 编译

C++ 是一种编译语言。这意味着要想使一个程序可以运行，你首先必须将它从人类可读的格式转换为机器可以“理解”的东西。这个转换过程由一个称为编译器的程序来做。你可以读或写的称为源代码或程序文本，计算机可以执行的称为可执行代码、目标代码或机器代码。典型的 C++ 源代



码文件的后缀为 .cpp(例如 hello_world.cpp)或 .h(例如 std_lib_facilities.h), 目标代码文件的后缀为 .obj(在 Windows 中)或 .o(在 UNIX 中)。代码一词是模棱两可的并且会引起混淆, 注意只有在可以明确表达含义时才使用它。除非特别说明, 否则我们用代码来表示“源代码”或“不包含注释的源代码”, 这是由于注释只是供人类阅读的, 在编译器生成目标代码时不会看到它。

编译器会阅读你的源代码, 并且尽力理解你所写的内容。编译器会检查你的程序在语法上是否正确, 每个单词是否已明确定义, 在程序中是否有不必实际执行就可以检测到的明显错误。你会发现编译器在语法上相当挑剔。忽略程序中的有些细节(例如#include 文件、分号或大括号)将会引起错误。与此类似, 编译器绝对不会容忍拼写错误。我们将通过一系列例子来解释这些, 在每个例子中有一个小错误。每个错误是我们经常犯的一类错误的例子:

```
// no #include here
int main()
{
    cout << "Hello, World!\n";
    return 0;
}
```

我们没有包括任何文件以告诉编译器 cout 是什么, 因此编译器会抱怨。为了纠正这个错误, 增加一个头文件:

```
#include "std_facilities.h"
int main()
{
    cout << "Hello, World!\n";
    return 0;
}
```

不幸的是, 编译器再次抱怨: 我们拼写错了 std_lib_facilities.h。编译器也不支持这样:

```
#include "std_lib_facilities.h"
int main()
{
    cout << "Hello, World!\n";
    return 0;
}
```

我们没有用一个"来终止字符串。编译器也不支持这样:

```
#include "std_lib_facilities.h"
integer main()
{
    cout << "Hello, World!\n";
    return 0;
}
```

在 C++ 中使用缩写 int 而不是单词 integer。编译器也不支持这样:

```
#include "std_lib_facilities.h"
int main()
{
    cout < "Hello, World!\n";
    return 0;
}
```

我们使用 < (小于操作符) 而不是 << (输出操作符)。编译器也不支持这样:

```
#include "std_lib_facilities.h"
int main()
{
    cout << 'Hello, World!\n';
    return 0;
}
```

我们使用单引号而不是双引号来限制字符串。最后，编译器发现这样的错误：

```
#include "std_lib_facilities.h"
int main()
{
    cout << "Hello, World!\n"
    return 0;
}
```

我们忘记使用分号来终止输出语句。需要注意的是，很多 C++ 语句是用一个分号(;)来终止的。编译器通过这些分号来识别一个语句在哪里终止，以及另一个语句从哪里开始。这里没有简短的、完全正确的、非技术方式的有关哪里需要分号的总结。现在，我们只是复制自己的应用模式，它可以归纳为：在每个没有由右侧大括号(})表示结束的表达式后面放置一个分号。

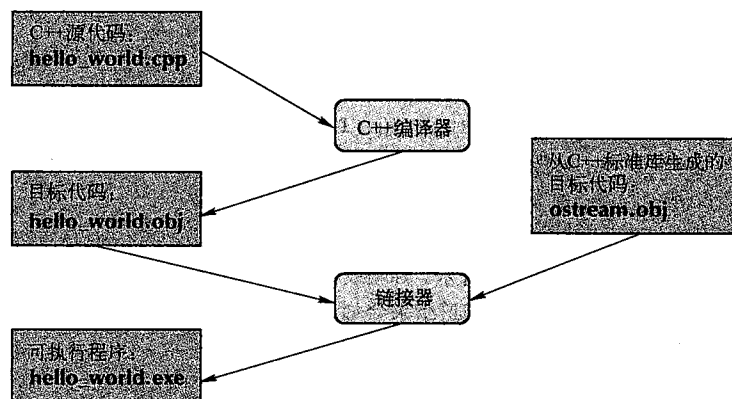
为什么我们要花费篇幅和你宝贵的时间给你看这些带有琐碎错误的小程序的例子呢？像所有的程序员一样，你会花费大量时间在程序源文本中查找错误。在大多数时候，我们看到的是含有错误的文本。毕竟，如果我们确信一些代码是正确的，那么我们通常会看其他代码或将时间用在别的地方。这令早期的计算机先驱们非常惊讶，他们曾经不得不花费自己的绝大部分时间来发现编程时产生的错误。这也令大多数的编程新手感到惊讶。

当你编程时，有时会对编译器感到相当懊恼。有时候，编译器会抱怨无关紧要的细节（例如缺少一个分号），或者一些你认为“明显正确”的东西。但是，编译器通常是正确的：当它给出一个错误消息并拒绝为你的源代码生成目标代码时，在你的程序中确实有不正确的地方，这意味着你写的程序不符合 C++ 标准的精确定义。

编译器没有常识（它不是人类），它对细节是非常挑剔的。由于编译器没有常识，因此你不能让它尝试着去猜测那些“看起来正确”但是不符合 C++ 定义的代码所表达的意思。如果你这样做并且编译器的猜测与你不同，这时你需要花费很多时间找出为什么程序没有按你的要求工作。当所有的事都说清和做到后，编译器会帮你从大量自己造成的问题中解脱出来。编译器将我们从问题中拯救出来，而不是引起问题。因此，请大家记住，编译器是你的朋友，编译器可能是你在编程时最好的朋友。

2.4 链接

程序通常由几个单独的部分组成，它们经常由不同的人来开发。例如，“Hello, World!”程序包含我们编写的部分和 C++ 标准类库。这些单独的部分（有时称为翻译单元）必须被编译，其目标代码必须被链接起来以形成一个可执行程序。用于将这些部分链接起来的程序通常称为链接器。



请注意目标代码和可执行程序是不能在系统之间移植的。例如，当你为一台 Windows 机器进行编译时，你得到的支持 Windows 的目标代码无法在 Linux 机器上运行。

库是一些代码的集合，它们通常是由其他人编写的，我们用#include 文件中的声明来访问这些代码。声明用于指出一段程序如何使用一条语句，我们将在后面的章节(如 4.5.2 节)中详细介绍声明。

由编译器发现的错误称为编译时错误，由链接器发现的错误称为链接时错误，直到程序运行时仍未发现的错误称为运行时错误或逻辑错误。通常来说，编译时错误比链接时错误更容易理解和修正，链接时错误比运行时错误和逻辑错误更容易发现和修正。在第 5 章中，我们将详细讨论这些错误和它们的解决方式。

2.5 编程环境

我们使用编程语言来编写程序。我们使用编译器将自己的源代码转换成目标代码，使用链接器将我们的目标代码链接成一个可执行程序。另外，我们使用一些程序在计算机中输入源代码文本并且编辑它。这些是最初的和最重要的工具，它们构成程序员的工具集合或“程序开发环境”。

如果你使用的是命令行窗口，就像很多专业程序员所做的那样，你将不得不自己来编写编译和链接命令。如果你使用 IDE(“交互式开发环境”或“集成式开发环境”)，就像很多程序员所做的那样，简单地点击正确按钮就可以完成这个工作。附录 C 介绍了如何在你的 C++ 实现中编译和链接。

IDE 通常包括一个具有有用特性的编辑器，例如用不同颜色的代码来区分你的源代码中的注释、关键字和其他部分，以及其他帮助你来调试代码、编译和运行代码的功能。调试是发现程序中的错误和排除错误的活动，你在前进的道路上会听到很多有关它的内容。

在本书中，我们使用微软的 Visual C++ 作为编程开发环境实例。如果我们简单地说“编译器”或是“IDE”的某些部分，那就是所指 Visual C++ 系统。但是，你可以使用一些提供最新的、符合标准的 C++ 实现的系统。我们所说的大多数内容(经过微小的修改)对所有的 C++ 实现都将是正确的，并且其代码可以在任何地方运行。在工作中，我们使用几种不同的实现。

简单练习

迄今为止，我们讨论了很多有关编程、代码和工具(例如编译器)的内容。现在，你可以运行一个程序。这是本书中和学习编程过程中的一个重点。这是你培养实践技能和好的编程习惯的开始。本章练习的重点在于使你熟悉软件开发环境。当你运行“Hello, World!”程序时，你将通过成为程序员的第一个重要的里程碑。

这个简单练习的目的是建立或加强你的实际编程技能，以及为你提供编程环境工具方面的经验。典型的简单练习是对一个独立程序的一系列修改，将琐碎的程序“发展”成一个实际程序的有用的部分。一套传统的练习是用于测试你的主动性、灵活性或创造性的。与此相反，简单练习很少需要你发挥创造力。典型的情况是顺序很关键，每个单独的步骤可能是容易(或琐碎)的。请不要自认为聪明地试图跳过某些步骤，这样通常会减慢你的进展或使你感到迷惑。

你可能会认为自己已经理解了阅读过的和导师或辅导员告诉你的任何事，但是重复和实践对提高编程能力是很必要的。在这方面，编程和体育、音乐、舞蹈或任何基于技能的行业是相似的。请想象人们试图经过常规练习就能在这类领域中参与竞争，你知道结果会如何。坚持练习对专业人员意味着终身的长期练习，是发展和维持一种高水平的实用技能的唯一方式。

因此，不要跳过这个简单练习，即使你多么想跳过去，它们从本质上来说是学习的过程。现在，从第一步开始并继续下去，测试每个步骤以确保你做得正确。

如果你不理解所使用的语法的细节也不必担忧,不要害怕向辅导员或朋友寻求帮助。坚持完成所有的简单练习和部分的习题,所有一切都会在适当的时候变得清晰。

那么,这是你的第一个简单练习:

1. 查看附录 C 并按照这些步骤的要求建立一个工程。建立一个名为 `hello_world` 的空白的 C++ 工程控制台。
2. 输入 `hello_world.cpp`, 完全按照下面的要求,将它保存在你的练习目录中,并将它包含在你的 `hello_world` 工程中。

```
#include "std_lib_facilities.h"
int main()    // C++ programs start by executing the function main
{
    cout << "Hello, World!\n"; // output "Hello, World!"
    keep_window_open();       // wait for a character to be entered
    return 0;
}
```

在一些 Windows 机器中需要调用 `keep_window_open()`, 以防止在你在有机会阅读输出之前窗口被关闭。这是 Windows 的一个特点,而不是 C++ 的。我们在 `std_lib_facilities.h` 中定义 `keep_window_open()`, 以便简化简单文本程序的编写。

你如何找到 `std_lib_facilities.h`? 如果你在上课,你可以问辅导员。否则,你可以从我们的支持站点 www.stroustrup.com/Programming 下载它。但是,如果你既没有辅导员又无法访问网站?(只有)在这种情况下,用下列语句代替 `#include` 语句:

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <cmath>
using namespace std;
inline void keep_window_open() { char ch; cin >> ch; }
```

这里直接使用了标准库,它将会跟随你直到第 5 章,并将在 8.7 节中详细解释。

3. 编译和运行“Hello, World!”程序。有些地方很可能不会正常工作。很少有人在初次尝试时使用一种新的编程语言或一个新的编程环境时就能成功。找到问题并修改它!这里的关键是向一个有经验的人寻求帮助,但是你要确定能够理解你看到的东西,这样你在进一步处理之前可以完全自己做。
4. 现在,你可能遇到一些错误并不得不纠正它。现在是更好地熟悉编译器的错误发现和错误报告功能的时候了!尝试来自 2.3 节的 6 个错误,以便查看编程环境对它们的反应。考虑至少 5 个可能发生的错误,它们是你在输入程序时造成的(例如忘记 `keep_window_open()`, 在输入单词时按下 Caps Lock 键,或者将分号输入成逗号),并查看在编译和运行每种错误时会发生什么?

思考题

这些思考题的基本思想是给你一个机会,以查看你是否注意到和理解了本章的关键点。在回答问题时你可能需要返回正文中,这是正常的和可以预料的。你可能需要重新阅读整个章节,这也是正常的和可以预料的。但是,如果你需要重新阅读整个章节或对每道思考题都有问题,那么你可能需要考虑自己的学习方式是否有效。你是否阅读得过快?你可能要停下来并做一下“试一试”吗?你会和朋友一起学习以讨论正文解释方面的问题吗?

1. “Hello, World!”程序的目的是什么?
2. 函数的 4 个部分的名字。
3. 函数命名必须出现在每个 C++ 程序中。
4. 在“Hello, World!”程序中, `return 0`; 这行的目的是什么?
5. 编译器的目的是什么?
6. `#include` 语句的目的是什么?
7. 文件名后缀为 `.h` 在 C++ 中表示什么?
8. 链接器为你的程序做什么?

9. 源文件和对象文件之间的区别是什么?
10. IDE 是什么以及它能为你做什么?
11. 如果你理解教材中的所有内容,为什么练习还是必要的?

大多数的思考题在它们出现的章节中有明确的回答。但是,我们偶尔会包含问题以提醒你在其他章节中有相关的信息,甚至有些内容可能不在本书范围内。我们认为这是正常的,更为重要的是编写好的软件和思考这样做的含义,而不是更适合于作为独立的章节或书籍出现。

术语

这些术语是编程和 C++ 方面的基本词汇。如果你希望理解人们谈到的关于编程的主题和阐明自己的思路,你应该知道每个术语的含义。

//	可执行程序	main()	<<	函数	目标代码
C++	头文件	输出	注释	集成开发环境 (IDE)	程序
编译器	#include	源代码	编译时错误	库	语句
cout	链接器				

你也可以以自己的语言逐步发展出一个词汇表。你可以通过重复每章后面的练习 5 来完成它。

习题

我们将简单练习与习题分别列出,在尝试做习题之前,请先完成本章中的简单练习。这样做将会节约你的时间。

1. 修改程序以输出下面 2 行
Hello, programming!
Here we go!
2. 扩展你曾经学习的知识,编写程序列出使计算机找到楼上浴室的指令(在 2.1 节中讨论)。你能讨论更多的人类可以假设而计算机不会的步骤吗?将它们加入你的列表。这是一个“像计算机一样思考”的好的开始。注意,对于大多数人来说,“去浴室”是一个完全充分的指令。对于那些对房子或浴室没有任何经验的人(想象一个石器时代的人被传送到你的餐厅),这个包含必要指令的列表可能会很长。请不要使用超过一页的指令。为了便于读者们阅读,你可以增加一个关于你所想象的房子布局的简短描述。
3. 编写一个有关如何从你的宿舍、公寓、房屋等的前门到你的教室(假设你在参观某个学校,如果你无法想象,选择其他目的地)前门的描述。请一个朋友尝试按照这些指令走,并且对他或她改进的路线加以解释。为了保持朋友关系,在将这些指令交给一个朋友之前进行“实地测试”是一个好的主意。
4. 找到一本好的菜谱。阅读有关烤制蓝莓松饼的指令(如果你在一个“蓝莓松饼”是一种陌生的、异国食品的国家,你可以用自己更熟悉的食品来代替)。请注意,在得到一点帮助和指令的情况下,这个世界上的大多数人都可以烤出美味的蓝莓松饼。这里不需要考虑高级的或困难的精细食品。但是,对于作者来说,本书中很少有习题像这个这样困难。只是通过一点儿练习,你所能做的事令人吃惊。
 - 重新写这些指令使每个单独的动作位于它们自己编号的段落中。认真列出每个步骤使用的所有原料和厨房用具。注意关键的细节,例如理想的烤箱温度、预热烤箱、准备烘烤的薄饼、烹调技术的方式,以及将松饼从烤箱中取出时的注意事项。
 - 从一个烹调初学者(如果你不是,请你认识的不会烹调的朋友帮忙)的角度来考虑这些指令。填写菜谱作者(几乎可以肯定是一个有烹调经验的人)漏掉的很明显的步骤。
 - 建立一个包含使用过的术语的词汇表。(松饼平底锅是什么?如何预热?你认为“烤箱”意味着什么?)
 - 现在,烤制一些松饼和享用你的成果。
5. 为“术语”中的每个术语书写一个定义。首先尽力看你是否不看本章就可以做到,然后浏览本章以找到这些定义。你会发现现在你初次尝试定义和看完本书之后的定义之间有趣的区别。你可以参考一些合适的在线词汇表,例如 www.research.att.com/~bs/glossary.html。通过在查找之前书写自己的定义,你会加强自己从学习中获得的知识。如果你需要重新读某个部分以形成一个定义,这也可以帮助你来理解它。你可以自由使用自己的词汇来进行定义,并且按照你认为合理的细节来完成定义。在通常情况下,主要定义

后面跟着一个例子将是有帮助的。你可以将这些定义存储在一个文件中，这样你可以通过增加它们形成后面章节的“术语”部分。

附言

“Hello, World!”程序有多么重要？它的目的是使我们熟悉基本的编程工具。当接触一个新的工具时，我们通常做一个非常简单的例子，例如“Hello, World!”。在这种方式下，我们将自己所学的东西分为两个部分：首先，我们通过简单程序学习有关工具的基础知识，然后，我们在没有被工具影响的情况下学习更复杂的程序。同时学习工具和语言的难度比先学一种后学另一种要大得多。这种将复杂任务分解成一系列小的（更容易管理的）步骤的学习方式，并不仅仅局限于编程和计算机方面。它在生活中的大多数领域是普遍的和有用的，特别是在那些需要实用技能的领域。

第3章 对象、类型和值

“幸运只青睐有准备的人。”

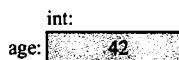
——Louis Pasteur

本章介绍程序中的数据存储和使用的基础知识。这样，我们首先关注从键盘读取数据。在建立起对象、类型、数值和变量的基本概念之后，我们介绍几种操作符，并且给出有关 `char`、`int`、`double` 和 `string` 类型的变量使用的例子。

3.1 输入

“Hello, World!”只是打印到屏幕。它产生输出，不读取任何内容，也就是不从用户那里得到输入。这令人相当厌烦。实际的程序通常基于我们给它的输入产生结果，而不是当我们每次执行它们时都做相同的事。

为了读取数据，我们需要将其读入某个地方，即我们需要在计算机内存中的某个地方放置读取的内容。我们将这样一个“地方”称为一个对象。一个对象是一个某种类型的内存区域，类型指定了可以放置什么样的信息。一个有名字的对象称为一个变量。例如，字符存放在 `string` 变量中，整数存放在 `int` 变量中。你可以将对象看成一个“盒子”，你可以在其中放置该对象类型的数值，如右图所示。



上图表示一个名为 `age` 的 `int` 类型的对象，其中保存的是整数值 42。通过使用一个字符变量，我们可以从输入中读取一个字符，然后将它打印出来，具体如下：

```
// read and write a first name
#include "std_lib_facilities.h"

int main()
{
    cout << "Please enter your first name (followed by 'enter'):\n";
    string first_name; // first_name is a variable of type string
    cin >> first_name; // read characters into first_name
    cout << "Hello, " << first_name << "!\n";
}
```

`#include` 与 `main()` 与第 2 章中的内容相似。由于我们的所有程序（直到第 12 章）都需要 `#include`，我们将不再介绍它以避免引起你分心。同样，我们有时需要将代码放入 `main()` 或其他函数中才能工作，就像下面这样：

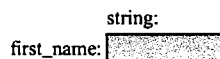
```
cout << "Please enter your first name (followed by 'enter'):\n";
```

我们假设你可以理解如何将这个代码加入一个完整的程序以便测试。

`main()` 中的第一行简单地输出一个信息，以便鼓励用户输入一个名字。这个信息通常称为提示符，这是由于它提示用户完成某个操作。下一行定义了一个名为 `first_name` 的 `string` 变量，接下来的程序读取键盘输入存入变量，然后输出一个欢迎词。接下来，我们逐行分析三行：

```
string first_name; // first_name is a variable of type string
```

本行会划分一个可以保存一个字符串的内存区域，并将它命名为 `first_name`，如右图所示。



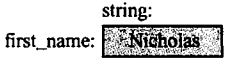
定义是用于将一个新的名字加入一个程序，并为一个变量分配内存空间的语句。

下一行将(键盘)输入的字符串读取到变量：

```
cin >> first_name; // read characters into first_name
```

名字 cin 是由标准库定义的标准输入流(读为“see-in”，“character input”的缩写)。操作符 >> 的第二个操作指定对象输入到哪里。因此，如果我们输入名字(例如 Nicholas)，后接一个换行，字符串 "Nicholas" 将会变成 first_name 的值，如右图

所示。新行是必要的以引起计算机的注意。



计算机简单地收集输入的字符，直到输入一个新行(按下回车键)。这段“延迟”给你改变主意的机会，在按回车键之前可以删除或修改某些字符。这个新行不会成为保存在内存中的字符串的一部分。

在将输入的字符串放入 first_name 之后，我们就可以使用它了：

```
cout << "Hello, " << first_name << "\n";
```

本行会在屏幕上打印 Hello，接着是 Nicholas(first_name 的值)，接着是! 和一个新行('\n')：

```
Hello, Nicholas!
```

如果我们喜欢重复和额外的输入，我们可以用三个单独的输出语句来代替：

```
cout << "Hello, ";
cout << first_name;
cout << "\n";
```

但是，我们不是打字员，更重要的是非常不喜欢不必要的重复(由于重复为出错提供了机会)，因此我们将三个输出语句合并为一个语句。

注意，我们在 "Hello," 处使用的是引号，而在 first_name 处没有这样做。我们希望输出字符串常量时使用引号。当我们不使用引号时，我们希望输出的是名字中的值。考虑一下：

```
cout << "first_name" << " is " << first_name;
```

在这里，"first_name" 输出的是十个字符 first_name，而 first_name 输出的是变量 first_name 中的值，在这种情况下是 Nicholas。因此，我们得到：

```
first_name is Nicholas
```

3.2 变量

如果没有存储在内存中的数据，我们基本上不能用计算机做任何有趣的事，正如上面的例子中所说的字符串输入。我们用来存储数据的“位置”称为对象。我们需要使用一个名字来访问一个对象。一个命名后的对象称为变量，它有特定的类型(例如 int 或 string)，类型决定我们将什么赋给对象(例如，123 可以赋给 int 型和 "Hello, World! \n" 可以赋给 string 型)，以及可以使用的操作(例如，我们可以对多个 int 型使用 * 操作符进行乘法运算，对多个 string 型使用 <= 操作符进行比较)。我们赋给变量的数据项称为值。一个用来定义变量的语句(通常)称为定义，一个定义可以(通常会)提供一个初始值。考虑一下：

```
string name = "Annemarie";
int number_of_steps = 39;
```

我们可以像这样来可视化这些变量：



我们不能将类型错误的值赋给一个变量：


```
string name2 = 39;           // error: 39 isn't a string
int number_of_steps = "Annemarie"; // error: "Annemarie" is not an int
```

编译器将会记录每个变量的类型，并确认你对它的使用是否与其类型一致，就像你在它的定义中所指定的那样。

C++ 提供了相当多的类型（参见 A.8 节）。但是，你只使用其中五种类型，就完全可以写出下面的好的程序：

```
int number_of_steps = 39;    // int for integers
double flying_time = 3.5;   // double for floating-point numbers
char decimal_point = '.';   // char for individual characters
string name = "Annemarie";  // string for character strings
bool tap_on = true;         // bool for logical variables
```

名字 double 的原因是历史的：double 是“双精度浮点”的简称。浮点是数学上实数概念在计算机中的近似。

注意，每种类型的文字常量都有自己的特殊风格：

```
39           // int: an integer
3.5          // double: a floating-point number
'.'          // char: an individual character enclosed in single quotes
"Annemarie"  // string: a sequence of characters delimited by double quotes
true         // bool: either true or false
```

一串数字（例如 1234、2 或 976）表示一个整数，在单引号中的一个字符（例如 '1'、'@' 或 'x'）表示一个字符，带小数点的一串数字（例如 1.234、0.12 或 .98）表示一个浮点值，在双引号中的一串字符（例如 "1234"、“Howdy!” 或 "Annemarie"）表示一个字符串。如果要获得文字常量的细节描述，参见 A.2 节。

3.3 输入和类型

输入操作 >>（“get from”）是对类型敏感的，它读取的值与变量类型需要一致。例如：

```
// read name and age
int main()
{
    cout << "Please enter your first name and age\n";
    string first_name;    // string variable
    int age;              // integer variable
    cin >> first_name;    // read a string
    cin >> age;           // read an integer
    cout << "Hello, " << first_name << " (age " << age << ")\n";
}
```

因此，如果你输入 Carlos 22，>> 操作符将 Carlos 读入 first_name，将 22 读入 age，并且生成这个输出：
Hello, Carlos (age 22)

为什么它不将 Carlos 22（全部）读入 first_name？这是由于按照规定，字符串的读取会被空白符所终止，包括空格、换行和 tab 字符。否则，空白符在默认情况下会被 >> 忽略。例如，你可以在读取的数字之前添加任意多的空格，>> 将会跳过它们并读取这个数字。

如果你输入 22 Carlos，你将看到奇怪的东西，直到你能够理解这一切。22 将会读入 first_name，这是由于 22 毕竟是一串字符。另一方面，Carlos 并不是一个整数，因此它不会被读取。这时的输出将是 22 和一些随机数，例如 -96739 或 0。为什么？因为你没有给 age 赋一个初始值，并且没能成功地读取一个值存入它。因此，当你开始执行时，就会得到内存中的某个部分的“垃圾值”。在 10.6 节中，我们讨论“输入格式错误”的处理方式。现在，我们只是初始化 age，这样在输入错误时，我们会获得一个可预测的值：

```
// read name and age (2nd version)
int main()
{
    cout << "Please enter your first_name and age\n";
    string first_name = "???"; // string variable
                                // ("???" means "don't know the name")
    int age = -1; // integer variable (-1 means "don't know the age")
    cin >> first_name >> age; // read a string followed by an integer
    cout << "Hello, " << first_name << " (age " << age << ")\n";
}
```

现在, 输入 22 Carlos 将会输出:

Hello, 22 (age -1)

注意, 我们可以在一个输入语句中读取几个值, 就像我们可以在一个输出语句中写入几个值一样。注意, << 和 >> 都是对类型敏感的, 因此我们可以输出 int 型变量 age 和字符文字 '\n', 以及 string 型变量 first_name 和字符串文字 "Hello, "、" (age" 和 ") \n"。

使用 >> 读取的 String(默认情况下) 会被空格所终止, 也就是说, 它只能读取一个字。但是, 我们有时需要读取多个字。当然会有多种方法来解决这个问题。例如, 我们可以像这样来读取一个包括两个字的名字:

```
int main()
{
    cout << "Please enter your first and second names\n";
    string first;
    string second;
    cin >> first >> second; // read two strings
    cout << "Hello, " << first << ' ' << second << '\n';
}
```

我们简单地使用 >> 两次, 每次针对一个名字。当我们想输出多个名字时, 我们必须在它们之间插入一个空白符。

试一试 运行这个“名字和年龄”的例子。然后, 修改它以月份的形式输出年龄: 读取输入的年龄并(使用 * 操作符)乘以 12。将年龄读入一个 double 型的变量, 使 5.5 岁的孩子骄傲于他比 5 岁的孩子大。

3.4 运算和运算符

除了指定什么值可以存储在一个变量中之外, 变量类型还决定了我可以对它进行什么操作和它们表示什么。例如:

```
int count;
cin >> count; // >> reads an integer into count

string name;
cin >> name; // >> reads a string into name

int c2 = count+2; // + adds integers
string s2 = name + " Jr. "; // + appends characters

int c3 = count-2; // - subtracts integers
string s3 = name - "Jr. "; // error: - isn't defined for strings
```

通过“错误”, 我们认识到编译器拒绝程序对字符串进行减。编译器确切地知道哪种操作可以应用于哪种变量, 这样可以防止很多错误的发生。但是, 编译器不知道哪种操作对你有用, 因此它很高兴接受合法的操作, 即使它们在你看来可能是荒谬的。例如:

```
int age = -100;
```

很明显，你的年龄不能是一个负数，但是没有人会告诉编译器，因此它会为这个定义生成代码。
下表给出了一些常见的和有用的类型可以使用的操作符：

	bool	char	int	double	string
赋值	=	=	=	=	=
加			+	+	
连接					+
减			-	-	
乘			*	*	
除			/	/	
余数(模)			%		
递加 1			++	++	
递减 1			--	--	
加 n			+= n	+= n	
添加到结尾					+=
减 n			-= n	-= n	
乘并赋值			* =	* =	
除并赋值			/ =	/ =	
余数并赋值			% =		
从 s 读到 x	s >> x	s >> x	s >> x	s >> x	s >> x
从 x 写到 s	s << x	s << x	s << x	s << x	s << x
等于	==	==	==	==	==
不等于	!=	!=	!=	!=	!=
大于	>	>	>	>	>
大于或等于	>=	>=	>=	>=	>=
小于	<	<	<	<	<
小于或等于	<=	<=	<=	<=	<=

空白表示一个操作符不能直接用于一种类型(尽管可能有间接使用这种操作符的方式，见 3.7 节)。我们将在后面的内容中解释这些操作符。这里的关键是有很多有用的操作符，它们对相似的类型通常是相同的。

我们来介绍一个涉及浮点数的例子：

```
// simple program to exercise operators
int main()
{
    cout << "Please enter a floating-point value: ";
    double n;
    cin >> n;
    cout << "n == " << n
        << "\nn+1 == " << n+1
        << "\nthree times n == " << 3*n
        << "\ntwice n == " << n+n
        << "\nn squared == " << n*n
        << "\nhalf of n == " << n/2
        << "\nsquare root of n == " << sqrt(n)
        << endl;    // another name for newline ("end of line")
}
```

很明显，常见的数学操作有常见的表示法和含义，这点和我们在小学学到的知识一样。很自然，并不是我们想对一个浮点数做的任何事(例如得到它的平方根)都有相应的操作符。很多操作都表示为命名函数的形式。在这种情况下，我们使用标准库中的 `sqrt()` 来得到 `n` 的平方根：`sqrt(n)`。这种表示法与数学中相似。我们将会逐渐学习使用函数，并在 4.5 节和 8.5 节中讨论它们

的细节。

试一试 运行这个小程序。然后，修改它以读取一个 int 型，而不是一个 double 型。注意，`sqrt()` 不是针对 int 型定义的，因此将 `n` 赋值给一个 double 型并执行 `sqrt()`。另外，“练习”一些其他操作。注意，对于 int 型来说，`/` 是整除，`%` 是余数(模)，因此 `5/2` 等于 2 (而不是 2.5 或 3)，`5%2` 等于 1。对整数 `*`、`/` 和 `%` 的定义，保证两个正整数 `a` 和 `b` 可以得到 `a/b * a + a%b == a`。

字符串拥有更少的操作符，但在第 23 章中将看到足够多的命名操作。但是，所支持的操作符都可以按常规方式使用。例如：

```
// read first and second name
int main()
{
    cout << "Please enter your first and second names\n";
    string first;
    string second;
    cin >> first >> second;           // read two strings
    string name = first + ' ' + second; // concatenate strings
    cout << "Hello, " << name << '\n';
}
```

字符串 + 意味着连接，也就是说，当 `s1` 和 `s2` 是字符串时，`s1 + s2` 也是字符串，包含来自 `s1` 的字符后接来自 `s2` 的多个字符。例如，如果 `s1` 的值为 "Hello"，`s2` 的值为 "World"，那么 `s1 + s2` 的值为 "HelloWorld"。字符串比较操作特别有用：

```
// read and compare names
int main()
{
    cout << "Please enter two names\n";
    string first;
    string second;
    cin >> first >> second;           // read two strings
    if (first == second) cout << "that's the same name twice\n";
    if (first < second)
        cout << first << " is alphabetically before " << second << '\n';
    if (first > second)
        cout << first << " is alphabetically after " << second << '\n';
}
```

在这里，我们使用 `if` 语句来根据条件选择动作，该语句将在 4.4.4.1 节中详细介绍。

3.5 赋值和初始化

在很多方面，最有趣的操作符是赋值，表示为 `=`。它为一个变量赋予一个新的值。例如：

```
int a = 3;    // a starts out with the value 3
```

a: 

```
a = 4;        // a gets the value 4 ("becomes 4")
```

a: 

```
int b = a;    // b starts out with a copy of a's value (that is, 4)
```

a: 

b: 

```
b = a+5;    // b gets the value a+5 (that is, 9)
```

```
a: 4
b: 9
```

```
a = a+7;    // a gets the value a+7 (that is, 11)
```

```
a: 11
b: 9
```

最后一次赋值需要注意。首先，很明显 $=$ 并不意味着等于， a 不等于 $a+7$ 。它意味着赋值，也就是将一个新的值赋予一个变量。 $a = a+7$ 所做的事如下：

- 1) 首先，得到 a 的值，这里是整数 4。
- 2) 其次，将 7 和 4 相加，得到整数 11。
- 3) 最后，将整数 11 赋予 a 。

我们也可以通过字符串来说明赋值：

```
string a = "alpha";    // a starts out with the value "alpha"
```

```
a: alpha
```

```
a = "beta";           // a gets the value "beta" (becomes "beta")
```

```
a: beta
```

```
string b = a;          // b starts out with a copy of a's value (that is, "beta")
```

```
a: beta
```

```
b: beta
```

```
b = a+"gamma";         // b gets the value a+"gamma" (that is, "betagamma")
```

```
a: beta
```

```
b: betagamma
```

```
a = a+"delta";         // a gets the value a+"delta" (that is, "betadelta")
```

```
a: betadelta
```

```
b: betagamma
```

以上，我们使用“以…开始”和“获得”来区别两种相似的操作，但两者在逻辑上是有区别的：

- 初始化(给一个变量它的初值)。
- 赋值(给一个变量一个新的值)。

这些操作是如此相似，因此 C++ 允许我们对它们使用相同的符号(=)：

```
int y = 8;             // initialize y with 8
x = 9;                  // assign 9 to x
```

```
string t = "howdy!";    // initialize t with "howdy!"
s = "G'day";            // assign "G'day" to s
```

但是，赋值和初始化在逻辑上是不同的。你可以通过类型描述(如 `int` 或 `string`)来区分它们，初始化总是从类型描述开始，而赋值并不需要这样做。从原则上来说，初始化时变量总是空的。另一方面，赋值在放入一个新的值之前，首先必须将旧的值清空。你可以将变量看做是一种小的盒子，值是一个可以放入其中的具体东西(例如一枚硬币)。在初始化之前盒子是空的，但是在初始化之后它总是包含一枚硬币，因此为了在里面放入一枚新的硬币，你(即赋值操作符)首先需要移

走旧的东西(“销毁旧的值”),而且你不能留下一个空盒子(必须赋予一个值)。在计算机内存中并不完全如此,但是它对于我们理解后面的内容没有坏处。

3.5.1 实例:删除重复单词

当我们想将一个新的值放入一个对象,就需要赋值操作。当你考虑赋值操作时,很明显它在多次重复做一些事情时赋值是最有用的。当我们想以一个不同的值重复做某事时,我们需要进行一次赋值。让我们来看一个小的程序,它在一连串单词中找到相邻的重复字符。这段代码是大多数的语法检查程序的一部分:

```
int main()
{
    string previous = " ";    // previous word; initialized to "not a word"
    string current;          // current word
    while (cin >> current) {  // read a stream of words
        if (previous == current) // check if the word is the same as last
            cout << "repeated word: " << current << "\n";
        previous = current;
    }
}
```

由于它没有告诉我们重复单词在文本中的哪个位置出现,因此这个程序对我们并不是很有帮助,但是现在它够用了。我们从如下一行开始逐行分析这个程序:

```
string current;    // current word
```

这是一个字符串变量,我们使用它来读取当前(即最近阅读)的单词:

```
while (cin >> current)
```

这个结构称为一个 while 语句,它对右侧的程序结构感兴趣,我们将在 4.4.2.1 节中详细介绍。while 的意思是当输入操作 `cin >> current` 成功的情况下, (`cin >> current`) 后面的语句将反复执行,而 `cin >> current` 成功的条件是从标准输入中读取字符。记住,对于一个 string, `>>` 读取的是用空格分开的单词。你可以通过给程序一个终止输入符号(通常是指文件结尾)来终止这个循环。在 Windows 系统的计算机中,使用 `Ctrl + Z` (同时按 Control 和 Z) 紧接着一个回车。在 UNIX 或 Linux 系统的计算机中,使用 `Ctrl + D` (同时按 Control 和 D)。

因此,我们所做的是读取一个单词到 `current`,然后将它与前一个单词(存储在 `previous` 中)比较。如果它们是相同的,我们将会:

```
if (previous == current)    // check if the word is the same as last
    cout << "repeated word: " << current << "\n";
```

然后,我们准备好对下一个单词重复进行上述操作。我们通过将 `current` 单词拷贝到 `previous` 中进行这个操作:

```
previous = current;
```

这可以处理我们开始后的所有情况。当第一个单词没有前一个单词可以比较时,这段代码将会如何处理呢?这个问题可以在定义 `previous` 时得到解决:

```
string previous = " ";    // previous word; initialized to "not a word"
```

" " 只包含一个字符(空格字符,通过按键盘中的空格键来得到)。输入操作符 `>>` 会跳过空格,我们不可能通过输入得到它。因此,第一次执行 while 语句时,检测

```
if (previous == current)
```

失败(正如我们所希望的)。

理解程序流程的一种方式“推演计算机的运行”,也就是按程序的顺序逐行执行指定的工作。在一张纸上画出很多方块,然后在里面写入程序运行的结果。按程序指定的方式修改储存在

其中的值。

试一试 你亲自用一张纸来执行这个程序。输入是“The cat cat jumped”。即使是有经验的程序员，当某段代码不那么清晰时，也会用这种技术来推演其结果。

试一试 运行“重复单词检测程序”。用句子“She she laughed He He He because what he did did not look very very good good”来测试它。这里有多少个重复的单词？为什么？在这里，单词的定义是什么？重复单词的定义又是什么？（例如，“She she”是否重复？）

3.6 组合赋值运算符

一个变量的递增（增加 1）在程序中很常用，C++ 为它提供了一个特定的语法。例如：

```
++counter
```

意味着

```
counter = counter + 1
```

这里有很多其他常用方式，可以基于变量的当前值来修改它。例如，我们可能想将它加 7、减 9 或乘 2。C++ 直接支持这些操作。例如：

```
a += 7;    // means a = a + 7
b -= 9;    // means b = b - 9
c *= 2;    // means c = c * 2
```

通常，对一些二进制操作符 oper，a oper = b 意味着 a = a oper b（参见附录 A.5）。首先，这一规则提供了操作符 +=、-=、*=、/= 和 %=。这提供了一种令人愉快的、紧凑的、直接反映我们观点的表示方法。例如，在很多应用领域中，/= 和 %= 被称为“缩放”。

3.6.1 实例：重复单词统计

考虑上面的检测重复的相邻单词的例子。我们可以通过得到重复的单词在序列中的位置来改进程序。我们可以设置一个简单的变量，简单地统计单词数并输出重复的单词数：

```
int main()
{
    int number_of_words = 0;
    string previous = " ";    // not a word
    string current;
    while (cin >> current) {
        ++number_of_words;    // increase word count
        if (previous == current)
            cout << "word number " << number_of_words
                << " repeated: " << current << "\n";
        previous = current;
    }
}
```

我们将单词计数器设置为 0。我们每次看到一个单词，就会将这个计数器递增：

```
++number_of_words;
```

这样，第一个单词变为数值 1，下一个单词变为数值 2，依次类推。我们也可以按以下方式完成相同功能

```
number_of_words += 1;
```

或者是

```
number_of_words = number_of_words + 1;
```

但是，++ number_of_words 更加简短，并且直接表达递增的思想。

注意，这个程序与 3.5.1 节中的程序是如此相似。很明显，我们只是将这个程序从 3.5.1 节拿来，并对它进行一点儿修改以实现我们的目标。这是我们解决一个问题时用到的一个非常通用

的技术：当遇到一个问题需要解决时，我们找到一个相似的问题并用我们的方案加以适当修改。不要从头开始，除非你不得不这样做。在一个程序早期版本的基础上修改通常会节省大量时间，我们将会从成功深入原始程序中受益良多。

3.7 命名

我们命名自己的变量，这样我们可以记住它们，并在程序的其他部分使用。在 C++ 中什么可以作为一个名字呢？在一个 C++ 程序中，一个名字必须以字母开始，并且只能包含字母、数字和下划线。例如：

```
x
number_of_elements
Fourier_transform
z2
Polygon
```

以下这些不是名字：

```
2x           // a name must start with a letter
time$to$market // $ is not a letter, digit, or underscore
Start menu   // space is not a letter, digit, or underscore
```

当我们说“不是名字”时，我们的意思是 C++ 编译器不认为它们是名字。

如果你阅读系统代码或机器生成的代码，你可能看到以下划线开始的名字，例如 `_foo`。你自己不要这样写，这样的名字是为编译器和系统实体保留的。尽量避免使用下划线，这样你的名字将不会与编译器生成的名字冲突。

名字是区分大小写的，也就是说，大写字母和小写字母是不同的，因此 `x` 和 `X` 是不同的名字。

下面这个小程序至少有 4 个错误：

```
#include "std_lib_facilities.h"

int Main()
{
    String s = "Goodbye, cruel world! ";
    cOut << s << '\n';
}
```

在定义名字时只用一个字符来区分通常不是一个好主意，例如 `one` 和 `One`，它不会使编译器混淆，但是它容易使程序员混淆。

试一试 编译“Goodbye, cruel world!”程序，并检查错误信息。编译器是否能发现所有错误？它遇到问题时的建议是什么？编译器是否混淆并发现超过 4 个错误？依次改正这些错误，首先从词法开始，看错误信息如何改变（与改进）。

C++ 语言保留了很多（大约 70 个）名字作为“关键字”。我们将在附录 A.3.1 中列出它们。你不能使用它们作为你的变量、类型、函数等的名字。例如：

```
int if = 7; // error: "if" is a keyword
```

你可以使用标准库中的名字（例如 `string`），但是你不应该这样做。如果你想要使用标准库的话，这样一个通用名字的重用将会带来麻烦：

```
int string = 7; // this will lead to trouble
```

当你为自己的变量、函数、类型等选择名字时，最好选择有特定含义的名字。也就是说，选择有助于人们理解你的程序的名字。如果你将变量胡乱命名为“简单型”的名字，例如 `x1`、`x2`、`s3` 与 `p7`，则你在理解程序要做什么时也会遇到问题。缩写和仅有首字母的缩写会使人糊涂，因此要谨

慎用。我们在编程时明白这些缩略语的含义，但是对于下面的名字，我们预料至少有一个使你感到困惑：

```
mtbf
TLA
myw
NBV
```

我们预料在几个月之内，我们自己也至少会忘掉其中一个名字的含义。

短名字(例如 `x` 与 `i`)在常规使用时是有意义的。也就是说，`x` 可能是一个本地变量或一个参数(参见 4.5 节与 8.4 节)，`i` 可能是一个循环的索引(见 4.4.2.3 节)。

不要使用很长的名字，它们难以输入，由于太长难以在一个屏幕中显示，也难以快速读取。下面这些名字可能是合适的：

```
partial_sum
element_count
stable_partition
```

下面这些名字可能太长：

```
the_number_of_elements
remaining_free_slots_in_symbol_table
```

我们的“风格”是在一个标识符中使用下划线来区分单词，例如 `element_count`，而不是其他方案(例如 `elementCount` 与 `ElementCount`)。我们不使用全部大写字母的名字，例如 `ALL_CAPITAL_LETTERS`，这是由于它们通常保留作为宏(参见 27.8 节与附录 A.17.2)，因此我们避免这样使用。我们使用首字母大写来定义自己的类型，例如 `Square` 与 `Graph`。C++ 语言和标准库不使用大写字母，因此它们使用 `int` 而不是 `Int`，使用 `string` 而不是 `String`。因此，我们的约定会帮助你减少对自己类型和标准类型的混淆。

避免使用容易错误、误读或混淆的名字。例如：

Name	names	nameS
foo	f00	f1
f1	fi	fi







字符 0、o、O、1、l 和 I 容易引起麻烦。

3.8 类型和对象

类型的概念是 C++ 和大多数编程语言的核心。让我们以更紧密和稍微带有技术性的观点来看待类型，特别是我们在计算过程中用来存储数据的对象类型。它将会在长时间运行时节省时间，它也可能避免引起你的混淆。

- 类型定义一组可能的值与一组操作(对于一个对象)。
- 对象是用来保存一个指定类型值的一些内存单元。
- 值是被解释为一个类型的内存中的一组比特。
- 变量是一个命名过的对象。
- 声明是命名一个对象的一条语句。
- 定义是为一个对象分配内存空间的声明。

我们可以非正式地将一个对象看做一个盒子，我们可以将指定类型的值放入这个盒子。一个 `int` 盒子可以保存整数，例如 7、42 与 -399。一个 `string` 盒子可以保存字符串值，例如 "Interoperability"、"tokens: ! @ # \$ % ^ & * " 与 "Old McDonald had a farm"。我们可以生动地将它想象为：

<code>int a = 7;</code>	a: 
<code>int b = 9;</code>	b: 
<code>char c = 'a';</code>	c: 
<code>double x = 1.2;</code>	x: 
<code>string s1 = "Hello, World!";</code>	s1: 
<code>string s2 = "1.2";</code>	s2: 

由于 string 要跟踪它保存的字符数, 因此 string 比 int 的表示方法更复杂。注意, 一个 double 保存一个数字, 而一个 string 保存多个字符。例如, x 保存数字 1.2, 而 s2 保存三个字符 '1'、'.' 与 '2'。字符的单引号和字符串常量并不保存。

每个 int 的大小是相同的。也就是说, 编译器为每个 int 分配相同的固定大小的内存。在一个典型的台式电脑中, 这个大小是 4 个字节(32 个比特)。与此类似, bool、char 与 double 是固定大小的。在通常情况下, 你会发现台式电脑为一个 bool 或一个 char 分配 1 个字节(8 个比特), 为一个 double 分配 8 个字节。注意, 不同类型的对象使用不同大小的空间。特别地, 一个 char 比一个 int 占用更少的空间, string 不同于 double、int 与 char, 不同大小的字符串占用不同大小的空间。

在内存中比特的含义完全依赖于访问它时所用的类型。我们这样考虑: 计算机内存不知道我们的类型, 只是将它保存起来。只有当我们决定内存如何解释时, 在内存中的比特才有意义。这个过程与我们每天使用数字相似。12.5 的含义是什么? 我们并不知道。它可以是 \$ 12.5、12.5cm 或 12.5gallons。只有当我们使用单位时, 才会定义 12.5 的含义。

例如, 当值 120 表示的是一个 int, 而值 'x' 表示的是一个 char 时, 它们在内存中的比特值相同。如果我们将它看成一个 string, 它将不会有意义, 并在我们试图使用它时出现运行错误。我们可以像下面这样生动地解释它, 使用 1 和 0 表示内存中的比特值:

00000000 00000000 00000000 01111000

这是一个内存区域(一个字)中的一组比特, 它们可以被读取为一个 int(120) 或一个 char('x', 只看最右侧的 8 个比特)。一个比特是计算机中的一个内存单元, 它可以保存一个值 0 或 1。想理解二进制数的含义, 见 A.2.1.1 节。

3.9 类型安全

每个对象在定义时被分配一个类型。对于一个程序或程序的一个部分, 如果使用的对象符合它们规定的类型, 那么它们是类型安全的。不幸的是, 有很多执行操作的方式不是类型安全的。例如, 在一个变量没有初始化之前使用它, 则认为不是类型安全的:

```
int main()
{
    double x;           // we "forgot" to initialize:
                        // the value of x is undefined
    double y = x;       // the value of y is undefined
    double z = 2.0+x;   // the meaning of + and the value of z are undefined
}
```

当使用没有初始化的 x 时, 一个实现甚至被允许出现一个硬件错误。记得初始化你的变量! 这个规则的例外只有很少(非常少), 例如我们立即将一个变量作为输入操作的目标, 但是记住初始化变量是一个好习惯, 它会为我们减少很多的麻烦。

完全的类型安全是最为理想的，因此它对语言来说应是一般规则。不幸的是，C++ 编译器不能保证完全的类型安全，但是通过良好的编码训练和运行时检查，我们可以避免违反类型安全。理想状态是永远不要使用编译器也不能保证是安全的语言功能：静态类型安全。不幸的是，这对于大多数有趣的编程应用过于严格。一种显然的退而求其次的方法是，由编译器隐式生成代码，检查是否有违反类型安全的情况并捕获它们，但这已超出了 C++ 的能力。当我们决定做(类型)不安全的事时，我们必须自己做相应的检查工作。当我们在本节中遇到这种情况时，我们将会指出来。

类型安全的思想在编写代码时非常重要。这是我们在前面章节花费时间介绍它的原因。请注意陷阱并避开它们。

3.9.1 安全类型转换

在 3.4 节中，我们发现不能直接将 char 相加，或者将一个 double 与一个 int 比较。但是，C++ 提供间接方式来完成这些操作。在必要时，一个 char 可以转换成一个 int，而一个 int 也可以转换成一个 double。例如：

```
char c = 'x';
int i1 = c;
int i2 = 'x';
```

这里的 i1 和 i2 都被赋值为 120，它是字符 'x' 在大多数常见的 8 比特字符集(例如 ASCII)中的整数值。这是一个简单和安全的方法，通过它可以获得一个字符的数字表示。由于没有信息丢失，我们称这种 char-int 的转换为安全的。也就是说，我们可以将 int 结果拷贝到一个 char 中，并且得到原始的值：

```
char c2 = i1;
cout << c << ' ' << i1 << ' ' << c2 << '\n';
```

这里将会打印

```
x 120 x
```

在这种情况下，一个值总是被转换成一个等价的值，或者一个最接近等价的值(对于 double)，那么这些转换就是安全的：

```
bool 到 char
bool 到 int
bool 到 double
char 到 int
char 到 double
int 到 double
```

最常用的转换是从 int 到 double，这是由于它允许在表达式中混合使用 int 和 double：

```
double d1 = 2.3;
double d2 = d1+2;    // 2 is converted to 2.0 before adding
if (d1 < 0)           // 0 is converted to 0.0 before comparison
    error("d1 is negative");
```

对于一个确实很大的整数，当它被转换成 double 时，我们(在有些计算机中)可能承受一些精度上的损失。这种情况不是很常见。

3.9.2 不安全类型转换

安全的转换对程序员通常是一个福音，它可以简化编写代码的过程。不幸的是，C++ 也允许(隐式的)不安全转换。所谓的不安全，我们指的是一个值可以转换成一个其他类型的值，这个值不等于原始的值。例如：

```
int main()
{
    int a = 20000;
    char c = a;    // try to squeeze a large int into a small char
    int b = c;
    if (a != b)    // != means "not equal"
        cout << "oops!: " << a << "!=" << b << '\n';
    else
        cout << "Wow! We have large characters\n";
}
```

这种转换又称为“缩小”转换，这是由于它们将一个值放入一个对象，而这个对象大小难以存放这个值。不幸的是，只有少数编译器会警告将 char 初始化为 int 的不安全。问题是一个 int 通常比一个 char 大，因此(在这种情况下)它可以保存一个 int 值，但是这个值并不能表示为一个 char。尝试执行这个程序，查看你计算机中的值 b(常见的结果是 32)。更进一步，完成实验：

```
int main()
{
    double d = 0;
    while (cin >> d) {    // repeat the statements below
                        // as long as we type in numbers
        int i = d;        // try to squeeze a double into an int
        char c = i;       // try to squeeze an int into a char
        int i2 = c;       // get the integer value of the character
        cout << "d==" << d    // the original double
             << " i==" << i    // converted to int
             << " i2==" << i2  // int value of char
             << " char(" << c << ")\n"; // the char
    }
}
```

我们使用 while 语句允许尝试很多值，这个语句将在 4.4.2.1 节中解释。

试一试 输入各种各样的值来运行这个程序。尝试小的值(例如 2 和 3)；尝试大的值(大于 127 和大于 1000)；尝试负值；尝试 56；尝试 89；尝试 128；尝试非整数值(例如 56.9 和 56.2)。另外，如何在你的机器中从 double 转换成 int，以及如何从 int 转换成 char。本程序将显示，对一个给定的整数值，你的机器将打印什么字符(如果存在的话)。

你将发现很多输入值产生“不合理”的结果。基本上，我们是在尝试将 1 加仑水倒入容量为 1 品脱的桶中(大约是将 4 升水倒入一个 500 毫升的杯子)。

```
double 到 int
double 到 char
double 到 bool
int 到 char
int 到 bool
char 到 bool
```

所有这些转换被编译器接受，即使它们是不安全的。所谓不安全是指它们保存的值可能与被赋予的值不同。为什么这会是个问题？这是由于我们经常不会怀疑一个不安全的转换会发生。考虑：

```
double x = 2.7;
// lots of code
int y = x;    // y becomes 2
```

在我们定义 y 时可能忘记 x 是一个 double，或者我们临时忘记 double 到 int 转换会截短(总是去掉小数点后的尾数)，而不是使用常用的四舍五入。发生的事情是完全可以预测的，但是在 int y = x; 处没有任何东西能提醒我们信息(.7)被丢掉了。

从 int 到 char 的转换不会出现截短的问题, int 和 char 都不能表示一个整数的一部分。但是, 一个 char 只能保存非常小的整数值。在一台 PC 机中, 一个 char 占用 1 个字节, 而一个 int 占用 4 个字节, 如右图所示。因此, 我们不能将一个大的数 (例如 1000) 放入一个 int 而不丢失任何信息: 这个值是“缩小的”。例如:



```
int a = 1000;
char b = a;    // b becomes -24 (on some machines)
```

不是所有的 int 值都有等价的 char, 而 char 值的确切范围依赖于特定的实现。在一台 PC 机中, char 值的范围是 $[-128; 127]$, 但是只有 $[0; 127]$ 可以方便地移植。这是由于并不是每台计算机都是 PC 机, 不同计算机的 char 值的范围不同, 例如 $[0; 255]$ 。

为什么人们接受缩小转换的问题? 主要原因是历史性的: C++ 从它的前辈语言 C 继承了缩小转换, 因此从 C++ 出现时很多代码就依赖于缩小转换。很多这种转换实际上不会引起问题, 这是由于它们所涉及的值碰巧在范围内, 并且很多程序员反对编译器“告诉它们做什么”。特别是对有经验的程序员来说, 这些不安全转换问题在小的程序中是可管理的。它们在大的程序中可能是错误的来源, 并且是一个新程序员出现问题的重要原因。但是, 编译器可以对多数的缩小转换发出警告。

如果你认为转换可能导致一个错误值, 那么你需要做什么? 在我们做本节中的第一个例子时, 你在赋值之前要简单检查这个值。5.6.4 节与 7.5 节介绍做这种检查的简单方式。

简单练习

在完成这个练习的所有步骤之后, 运行你的程序以确认它确实在做你希望它做的事。列出那些曾经出现的错误, 这样以后可以尽量避免它们。

1. 这个练习是编写一个程序, 基于用户输入生成一封简单格式的信。首先, 输入来自 3.1 节的代码, 提示用户输入他或她的名字, 并且输出“Hello, first_name”, 这里的 first_name 是用户输入的名字。然后, 按以下要求修改你的代码: 将提示修改为“Enter the name of the person you want to write to”, 并将输出修改为“Dear first_name,”。不要忘记逗号。
2. 增加一行或两行前言, 例如“How are you? I am fine. I miss you.”确定首行需要缩进。增加由你选择的几行, 这是你的信。
3. 现在, 提示用户输入另一个朋友的名字, 并将它保存在 friend_name 中。在你的信中增加一行: “Have you seen first_name lately?”
4. 声明一个名为 friend_sex 的 char 变量, 并将它的值初始化为 0。如果这个朋友是男性, 提示用户输入一个 m; 如果这个朋友是女性, 提示用户输入一个 f。为变量 friend_sex 分配输入值。然后, 使用两个 if 语句完成以下输出:
如果这个朋友是男性, 输出“If you see friend_sex please ask him to call me.”。
如果这个朋友是女性, 输出“If you see friend_sex please ask her to call me.”。
5. 提示用户输入收信人的年龄, 并为它分配一个 int 变量 age。让你的程序输出“I hear you just had a birthday and you are age years old.”如果 age 小于等于 0 或大于等于 110, 调用 error(“you'r kidding!”)。
6. 在你的信中增加下面的内容:
如果你朋友的年龄小于 12, 输出“Next year you will be age + 1.”。
如果你朋友的年龄等于 17, 输出“Next year you will be able to vote.”。
如果你朋友的年龄大于 70, 输出“I hope you are enjoying retirement.”。
7. 添加“Yours sincerely,”接着是两个空行, 后面是你的名字。

思考题

1. 术语 prompt 的含义是什么?
2. 哪种操作符用于读取值并存入变量中?

3. 如果你希望用户在你的程序中为一个命名为 `number` 的变量输入一个整数值, 如何用两行代码来完成它并将值输入你的程序中?
4. `\n` 的名称是什么和它的目的是什么?
5. 怎样终止输入一个字符串?
6. 怎样终止输入一个整数?

7. 如何将你书写的

```
cout << "Hello, ";
cout << first_name;
cout << "\n";
```

作为一行代码来输入?

8. 对象是什么?
9. 文字常量是什么?
10. C++ 中有哪几种文字常量?
11. 变量是什么?
12. 一个 `char`、`int` 和 `double` 的典型大小是多少?
13. 我们用哪种方式测试内存中的小实体(例如 `int` 和 `string`)的大小?
14. 操作符 `=` 与 `==` 之间的区别是什么?
15. 定义是什么?
16. 什么是一次初始化, 它和一次赋值的区别是什么?
17. 什么是字符串连接, 如何使它在 C++ 中正确工作?
18. 在以下名字中, 哪些在 C++ 中是合法的? 如果一个名字是不合法的, 为什么?

<code>This_little_pig</code>	<code>This_1_is_fine</code>	<code>2_For_1_special</code>
<code>latest thing</code>	<code>the_\$12_method</code>	<code>_this_is_ok</code>
<code>MiniMineMine</code>	<code>number</code>	<code>correct?</code>

19. 请举出 5 个容易引起混淆、你不会使用的合法名字。
20. 选择名字的好规则有哪些?
21. 什么是类型安全, 为什么它是重要的?
22. 为什么从 `double` 转换成 `int` 是一件坏事?
23. 请定义一个协助判断从一种类型到另一种类型的转换是否安全的规则。



术语

赋值	定义	运算	<code>cin</code>	递增	运算符
连接	初始化	类型	转换	名字	类型安全
声明	缩小	值	递减	对象	变量



习题

1. 如果你还没有开始这样做, 请先做本章的“试一试”练习。
2. 编写一个 C++ 程序, 将英里转换成公里。你的程序应该有一个合理的提示, 要求用户输入一个表示英里的数字。提示: 这里是 1.609 公里要转换成英里。
3. 编写一个程序, 不做其他的任何事情, 只声明一系列合法与不合法的变量名(例如 `int double = 0;`), 这样你可以看到编译器的反应。
4. 编写一个程序, 提示用户输入两个整数值。将这些值保存在 `int` 变量 `val1` 和 `val2` 中。编写程序决定这些值中的最小值、最大值、和、差、乘积和比率, 并且将它们报告给用户。
5. 修改上面程序, 让用户输入浮点数值并将它们保存在 `double` 变量中。比较你选择的多种输入在两个程序的输出。这些结果是否相同? 它们是否对? 区别是什么?
6. 编写一个程序, 提示用户输入三个整数值, 然后按数值次序输出这些值并以逗号隔开。因此, 如果用户输入值为 10 4 6, 输出值为 4, 6, 10。如果有两个值相同, 那么将它们连续输出。因此, 输入 4 5 4 将会输出 4, 4, 5。

7. 重做练习 6, 但是输入 3 个字符串。因此, 如果用户输入“Steinbeck”、“Hemingway”和“Fitzgerald”, 输出将是“Fitzgerald, Hemingway, Steinbeck”。
8. 编写一个程序, 测试一个整数是奇数还是偶数。记住确认你的输出是清楚和完整的。换句话说, 不要只是输出“yes”或“no”。你的输出应该是独立的, 例如“The value 4 is an even number.”提示: 阅读 3.4 节中的余数(模)操作。
9. 编写一个程序, 将数字的英文单词(例如“zero”和“two”)转换成数字(例如 0 和 2)。当用户输入一个数字的英文拼写, 程序将打印出对应的数字。针对数值 0、1、2、3 和 4 完成这个操作, 如果用户输入无法对应的值(例如“stupid computer!”), 程序输出“not a number I know”。
10. 编写一个程序, 执行一个包括两个运算的操作, 然后输出结果。例如:

+ 100 3.14

* 4 5

将操作读入一个字符串称为 operation, 用一个 if 语句判断哪个操作是用户希望的, 例如 if (operation == "+")。将运算读入 double 类型的变量。实现这些称为 +、-、*、/ 的操作, 加、减、乘、除都有各自明显的意义。

11. 编写一个程序, 提示用户输入一美分(1 美分硬币)、五美分(5 美分硬币)、十美分(10 美分硬币)、二十五美分(25 美分硬币)、半美元(50 美分硬币)和一美元(100 美分硬币)的数量。对每种面值的硬币, 分别提示用户输入其数量, 例如“How many pennies do you have?”然后, 程序将输出类似下面的内容:

You have 23 pennies.

You have 17 nickels.

You have 14 dimes.

You have 7 quarters.

You have 3 half dollars.

你可能需要发挥想象力才能将合计值右对齐输出, 但请尽力尝试, 你可以完成它。然后做出一些改进: 如果某个面值只有一枚硬币, 确保输出语法是正确的, 例如, 应输出“14 dimes”和“1 dime”(而不是“1 dimes”)。另外, 将合计值用美元和美分来表示, 例如用 \$ 5.73 来代替 573 美分。



附言

请不要低估类型安全概念的重要性。类型是大多数正确程序的核心概念, 大多数用于构建程序的有效技术依赖于类型的设计与使用, 正如我们将在第 6 章、第 9 章、第二部分、第三部分、第四部分中看到的一样。

第4章 计 算

“如果不要结果的正确性，我可以让程序运行得任意快”。

——Gerald M. Weinberg

本章将介绍一些与计算相关的基本概念。我们将着重讨论如何通过一系列指令来计算一个数值量(表达式)；如何在可替代操作中进行选择(选择)；如何对一系列数据进行重复计算(迭代)。此外，我们还将介绍一种可以被单独划分出来的计算操作(函数)。本章内容的核心是通过介绍计算让读者能够编写出正确而且规范的程序。为了让读者更好地理解计算，我们将引入向量这一概念来表示数据序列。

4.1 计算

有一种观点认为，程序就是以计算为目的的，即程序都要有输入和输出。在这里，我们把能够运行程序的硬件设备称为计算机。如果我们用广义的概念来理解输入和输出的话，那么上述的观点可以认为是正确的。程序的输入来源有很多：可以是键盘、鼠标、触摸屏、文件、其他输入设备、其他程序，或者同一程序的其他部分。在这里，“其他输入设备”的范围很广，它表示了一大类实际输入设备：音乐键盘、摄像机、网络设备、温度传感器、数字图像传感器等。随着技术的进步，输入设备可以千变万化。



为了处理输入，程序通常包含一些数据，有时称为程序的数据结构或程序的状态。例如，日历年程序需要记录不同国家的公共假期和用户的事务安排表。这些数据一部分是在程序中设定好的，还有一部分是在程序运行期间，程序通过各种输入设备获取的。例如，通过用户的输入，日历年程序可以准确地建立用户的事务安排表。对于一个日历年程序来说，主要输入包括对日期的查询(一般通过鼠标点击)和对用户事务安排表的处理(通常使用键盘输入相关信息)。输出包括日历和事务安排表的显示、加上按钮和提示符显示等。

输入的来源非常广泛。同样，输出也有很多不同的途径：可以是屏幕、文件以及其他设备，或者其他程序，甚至可以是同一程序的其他部分。输出设备很多，例如网络接口、音乐合成器、电动马达、发光器和加热器等。

从编程的角度看，最重要也是最有趣的两类输入、输出是“从其他程序输入或输出”和“从同一程序的其他部分输入或输出”。本书后续的大部分内容可以视为后一种类型的实例：在协作完成一个大的软件时，应该如何合理地设计程序结构，并能够保证每一个子程序之间都能够正确地共享和交换数据？这是编程的核心问题，下图说明了这一过程：



其中，I/O是“input/output”的缩写。在图中，一部分代码的输出是下一部分代码的输入。而子程序之间的数据共享可以通过内存、非易失存储设备(例如硬盘)或者网络完成。在这里，“子

程序”是指程序中的各类函数，包括根据输入参数产生输出结果的函数（例如求浮点数的平方根函数），对物理对象进行操作的函数（例如在屏幕上画线的函数），或者修改现有程序数据表的函数（例如在顾客信息表中增加一个名字）。

通常意义上的“输入”和“输出”是指信息进入和离开计算机。正如前文所述，也可以将“输入”和“输出”引入到子程序的数据传递。通常，子程序的输入称为参数，子程序的输出称为结果。

从这个意义上来说，计算就是基于输入生成输出的过程。例如，基于参数 7（输入），计算过程 `square`（函数）可以得到结果 49（输出）（具体细节参见 4.5 节）。有趣的是，直到 20 世纪 50 年代，计算机的定义还是一个从事计算任务的人，例如会计、导航员或物理学家。今天，人类社会的大部分计算任务都是由各种类型的计算机（真正的机器）完成的。日常生活中最常见的就是计算器了。

4.2 目标和工具

程序员的任务就是将计算表示出来，并且做到：

- 正确
- 简单
- 高效

请注意上述顺序。一个输出错误结果的快速程序是没有任何意义的。同样，一个正确、高效但是非常复杂的程序，最终的结果往往是被放弃或者重写。记住，为了适应不同的需求和硬件环境，有用的程序总会被无数次改写。因此，一个程序，或者它的任一子程序，应该以尽可能简单的方式来实现。举个例子，假设你为学校的孩子写了一个非常棒的算术教学程序，而这个程序的内部组织非常糟糕。这个程序必须与孩子们交互，那你应该用什么语言呢？英语？英语和西班牙语？如果程序要在芬兰或科威特用，那又该怎么办呢？无疑，为了能适应不同地区的需要，程序使用的交互语言必须能够被修改。如果程序的结构是非常混乱的话，原本逻辑上很简单的（但实际中总是很困难的）修改操作会变成一项艰巨的任务。

在我们开始编写代码的时候，就要特别关注正确、简单和高效这三个基本原则，这也是专业程序员的最重要职责。在实际工作中，遵循这些原则意味着代码仅仅能用是不够的，我们必须认真考虑代码的结构。一个看似矛盾的事实是，关注代码结构和“代码质量”是程序取得成功的最快速途径。这一点很好理解，编程时对编码结构和质量所付出的努力，可以大大简化最令人沮丧的编程工作：调试。这是因为好的程序结构不但可以减少错误的发生，而且还能缩短发现并改正错误的时间。

程序的组织体现了程序员的编程思路，目前的手段主要是把一个大的计算任务划分为许多小任务。这一技术主要包括两类方法：

- 抽象：即不需要了解的程序具体实现细节被隐藏在相应的接口之后。例如，为了实现电话簿的排序，我们不需要了解排序算法的细节（已经有很多书讨论如何排序了）。我们要做的只是调用 C++ 标准库函数中的排序函数就可以了，即如何调用这一排序函数：编号 `sort(b, e)`，这里 `b` 和 `e` 分别指示电话簿的开始和结束。另一个例子是内存的使用，直接使用内存空间是一个糟糕的想法。通常，我们通过变量（参见 3.2 节）、标准库向量（参见 4.6 节，第 17 ~ 19 章）和映射（参见第 21 章）等来访问内存。
- 分治：把一个大问题分为几个小问题分别解决。例如，在建立一个字典的时候，可以把这一任务分解为三个子任务：读数据、数据排序和输出数据，每个子任务都明显小于原来的任务。

这么做有什么用呢？毕竟，由很多部分构成的程序要比一个完整的程序规模大。直接原因是大问题处理起来太困难，这种情况不只存在于程序设计，也存在于很多其他领域。对于这一情况，我们的解决办法是将问题不断分解、细化，直到问题小到能够被我们很好地理解和解决为止。以编程为例，一个 1000 行程序的规模是一个 100 行程序的 10 倍。但是 1000 行程序的错误会远超过 100 行程序的 10 倍。解决的办法就是把这个 1000 行程序分解为多个子程序，每个子程序不超过 100 行。对于大型软件来说，例如一个 1000 万行的程序，使用抽象和分治等技术就不只是一个编程建议了，而是必须这样做。编写并维护一个庞大的单一程序是很困难的。对于本书剩余的内容，读者不妨尝试利用已有的工具和方法，把一些大问题分解为一系列小问题。

在考虑划分一个程序前，我们首先要明确手里有哪些工具可以表示各个子程序及其之间的关系。这是因为一个能够提供充分的接口和功能的库可以大大简化程序的划分工作。凭空想象什么是程序的最优划分方法是不切实际的，按照功能对程序进行划分是目前最常用的方法。无疑，利用已有的各类程序库能够简化按功能进行程序划分的工作量。事实上，利用类似 C++ 标准库这种已有的库，不但可以减少编程的工作量，而且可以减少测试和写文档的工作量。例如，`iostreams` 库屏蔽了 I/O 设备的实现细节。程序员可以直接调用相应库函数，而不需要了解具体 I/O 接口是如何实现的，这就是抽象方法的一个具体实例。在后续章节中，我们将展示更多例子。

请注意我们对结构和组织的强调：要写出好的程序并不仅仅是堆砌大量的语句。我们为什么在此刻就提及这一点呢？毕竟这对于你（或者至少是大多数读者）来说有些早，你此时对代码还没有什么概念，写出其他人能用于实际生活的代码更是几个月后才可能的事情。我们之所以这么早就提及结构和组织的重要性，是想帮助你确立正确的学习重点。人们很容易禁不住诱惑，一头扎进具体的、能立刻显现出作用的程序设计技术（如本章剩余部分所介绍的那些内容），而忽略那些“软知识”，即软件开发艺术中基础概念部分。但好的程序员和系统设计者知道（通常是从惨痛的教训中学会的），对结构的关注是好软件的核心，而忽视结构会导致严重的混乱。没有结构，你就是在用泥砖盖房子。虽然泥砖也能盖房子，但绝不可能盖起五层高楼（因为泥砖没有足够的结构强度支撑这样的高楼）。如果你一心想构建能长久留存的东西，就应该在设计过程中对代码结构和组织投入更多的关注，而不是在发生错误后再被迫回过头来关注这些方面。

4.3 表达式

表达式是程序的最基本组成单元。表达式就是通过一系列操作来计算一个数值量。最简单的表达式是字面常量，例如 10、'a'、3.14 和 "Norah"。

变量名也是一种表达式，变量表示与名字对应的那个对象。例如：

```
// compute area:
int length = 20;           // a literal integer (used to initialize a variable)
int width = 40;
int area = length*width;    // a multiplication
```

在这里，字面常量 20 和 40 用于初始化变量 `length` 和 `width`。然后，`length` 和 `width` 进行乘法操作，即 `length` 和 `width` 所表示的值相乘。这时候，`length` 表示名字为 `length` 的对象的值。考虑如下情况：

```
length = 99; // assign 99 to length
```

该语句中的 `length` 位于赋值符号左边（即 `length` 是左值），其含义是名字为 `length` 的对象，因此赋值表达式的含义是“把 99 赋给名为 `length` 的对象”。要注意区分 `length` 用于赋值操作符左边和右边的含义是不同的，`length` 在左边时（即 `length` 是左值）表示“名为 `length` 的对象”，在右边时（即 `length` 是右值）表示“名为 `length` 的对象的值”。通过下面的图可以更清楚地解释这个概念：


```

int:
length: 99

```

上图表示了一个名为 length 的整型对象，其值为 99。当 length 是左值时，length 表示这个对象本身；当 length 是右值时，length 表示这个对象的值。

我们还可以使用运算符（如 + 和 ×）表示更复杂的表达式。如果需要的话，使用括号还可以构成复合表达式：

```
int perimeter = (length+width)*2; // add then multiply
```

如果没有括号的话，表达式必须表示为：

```
int perimeter = length*2+width*2;
```

显然，这种表示方法很繁琐，而且容易出错：

```
int perimeter = length+width*2; // add width*2 to length
```

这个语句的错误是语义错误而不是语法错误，编译器认为是合法的，该语句通过一个有效的表达式初始化 perimeter 变量。由于编译器不清楚 perimeter 的数学定义，因此编译器不能确定表达式是否有意义。

按照运算符的优先级规则，length + width * 2 表示 length + (width * 2)，同样，a * b + c/d 表示 (a * b) + (c/d)，而不是 a * (b + c)/d。A.5 节给出了运算符优先级表。

使用括号的第一条原则是“如果对运算符优先级不确定，就用括号”。当然，要尽量熟悉优先级规则，像 a * b + c/d 这种简单表达式还加括号的话，无疑会降低程序的可读性。

可读性为什么这么重要呢？因为程序是给人读的，读者可能是编程者本人，也可能是其他人。丑陋的代码不但会降低程序的可读性和可理解性，而且难以发现和改正程序的错误，因为丑陋的代码往往会导致难以发现其中的语义错误，难以让人相信它是正确的。记住，绝对不要在程序中使用非常复杂的表达式。例如：

```
a*b+c/d*(e-f/g)/h+7 // too complicated
```

另外，变量的命名应该有对应的含义。

4.3.1 常量表达式

程序中经常会用到常量。例如，一个与几何相关的程序会用到 pi，一个英寸到厘米的转换程序会用到转换系数 2.54。显然，常量名应该能够体现它的含义（例如，我们一般用 pi，而不是 3.14159）。而且，常量的值也不应该经常改变。因此，在 C++ 语言中提供了符号常量来表示那些在初始化后值就不再改变的数值量。例如：

```
const double pi = 3.14159;
pi = 7; // error: assignment to const
int v = 2*pi/r; // OK: we just read pi; we don't try to change it
```

常量对于维护程序的可读性具有重要作用。大部分人可能都知道 3.14159 表示的是 pi，但 299792458 表示什么就没有人能猜到了。进一步讲，如果要求程序把 pi 的精度提高到 12 位有效数字，则需要改变程序中所有用到 pi 的语句。一种可行的方法是搜索程序中所有包含 3.14 的地方，但对于使用 22/7 来代替 pi 的语句就搜索不到了。因此，最好的方法是在程序中只有一个定义 pi 的语句，其他用到 pi 的语句都使用该常量。需要修改 pi 值的时候，只修改 pi 的定义语句即可：

```
const double pi = 3.14159265359;
```

因此，我们的建议是：除了个别情况（例如 0 和 1），程序中应该尽量少用字面常量，而是尽可能地使用符号常量。在代码中，这种不能直接识别的字面常量通常称为魔数（magic constant）。

在一些情况下，例如在标号中（4.4.1.3 节），C++ 语言允许表达式与整数组合构成常量表达

式。例如：

```
const int max = 17; // a literal is a constant expression
int val = 19;
```

```
max+2 // a constant expression (a const int plus a literal)
val+2 // not a constant expression: it uses a variable
```

顺便说一下，299792458 是一个基本的物理常量：它是光在真空中的传播速度，单位是米/秒。当你不知道这一点的时候，在代码中看到这个字面常量肯定会犯糊涂。因此，要避免使用魔数。

4.3.2 运算符

到目前为止，我们用的都是最简单的运算符，接下来你将会看到许多复杂运算符的使用方法。后面我们将在遇到运算符时加以详细描述，大多数运算符都很好理解。下表给出了常用的运算符：

	名 称	说 明
f(a)	函数调用	a 做为函数 f 的参数
++lval	前增	变量值加 1
--lval	前级减	变量值减 1
!a	非	结果是布尔类型
-a	单目减	
a*b	乘	
a/b	除	
a%b	取模	仅用于整型
a+b	加法	
a-b	减法	
out << b	将 b 写到 out	out 是 ostream 对象
in >> b	从 in 中读取数据存到 b 中	in 是 istream 对象
a < b	小于	结果是布尔类型
a <= b	小于等于	结果是布尔类型
a > b	大于	结果是布尔类型
a >= b	大于等于	结果是布尔类型
a == b	等于	不要与赋值混淆
a != b	不等于	结果是布尔类型
a && b	逻辑与	结果是布尔类型
a b	逻辑或	结果是布尔类型
lval = a	赋值	不要与等于混淆
lval * = a	复合赋值	等价于 lval = lval * a, 类似还有 /、%、+、-

上表中的 lval 表示左值，即它可以出现在赋值号左边，在附录 A.5 中有详细介绍。

逻辑运算符 &&、|| 和 ! 的例子可以分别在 5.5.1 节、7.7 节、7.8.2 节和 10.4 节中找到。

需要注意的是，表达式 $a < b < c$ 表示 $(a < b) < c$ ， $a < b$ 的结果是布尔值：true 或 false。因此，表达式 $a < b < c$ 的值等于 $\text{true} < c$ 或者 $\text{false} < c$ ，而不是 $a < b < c$ 表示“b 的值是否介于 a 和 c 之间”。实际上，表达式 $a < b < c$ 是没有用处的，在进行比较操作时，千万不要写出这样的表达式。如果在别人的代码中发现了这种表达式，这往往意味着一个错误。

增量表达式至少有以下三种形式：

```
++a
a+=1
a=a+1
```

哪种方式比较好？为什么呢？建议使用第一种方式 ++a，它直观地表示了增量的含义，显示了我们要做什么（对 a 加 1），而不是怎么做（a 加 1，然后结果写到 a）。通常，我们认为能够更直接地

体现程序思想的编程方式更好一些,因为这种方式更准确,并且更容易被读者理解。假如使用 $a = a + 1$ 的话,读者可能会想,程序的原意真的是要对 a 加 1 吗?不是是要做 $a = b + 1$ 、 $a = a + 2$ 或者 $a = a - 1$ 但输入出错了吧!而使用 $++a$ 方式就不会引起这样的疑问。需要注意的是,上述只讨论程序的正确性和逻辑性,与程序的效率无关。实际上,目前的编译器对 $a = a + 1$ 和 $++a$ 的处理是一样的。同样,我们建议编程时使用 $a * = scale$ 而不是 $a = a * scale$ 。

4.3.3 类型转换

表达式中允许存在不同的数据类型。例如, $2.5/2$ 是一个 `double` 类型除以一个 `int` 类型。这表示什么呢?我们应该做整型除法还是双精度浮点型除法呢?整型除法时余数被丢弃,例如 $5/2$ 的结果为 2。浮点型除法时余数被保留,例如 $5.0/2.0$ 是 2.5。“ $2.5/2$ 是整型除法还是浮点型除法?”的答案是“浮点型除法,因为整型除法会丢失余数”。也就是说, $2.5/2$ 的结果是 1.25 而不是 1。我们遵循这样的规则:如果算术表达式中有 `double` 类型数据的话,就进行浮点型算术计算,结果为 `double` 类型;否则就使用整型算术计算,结果为 `int` 类型。例如:

$5/2$	结果是	2 (不是 2.5)
$2.5/2$	等同于	$2.5/\text{double}(2)$ 结果是 1.25
$'a' + 1$	等同于	$\text{int}('a') + 1$

这意味着编译器会把上述运算中的 `int` 自动转换为 `double`,或将 `char` 自动转换为 `int`。在运算完成的时候,这种转换也已经完成。例如:

```
double d = 2.5;
int i = 2;

double d2 = d/i; // d2 == 1.25
int i2 = d/i;    // i2 == 1
d2 = d/i;        // d2 == 1.25
i2 = d/i;        // i2 == 1
```

需要特别注意浮点运算表达式中的整数除法。例如,摄氏温度与华氏温度的转换公式为: $f = 9/5 * c + 32$, 程序代码如下:

```
double dc;
cin >> dc;
double df = 9/5*dc+32; // beware!
```

不幸的是,上述程序不能正确实现摄氏温度与华氏温度的转换功能,因为 $9/5$ 的值是 1 而不是我们期望的 1.8。如果要达到我们期望的结果,必须将 9 或 5 (或者二者)转换为 `double` 类型。

```
double dc;
cin >> dc;
double df = 9.0/5*dc+32; // better
```

4.4 语句

4.3 节介绍了利用各种运算符组成表达式来进行相应的数值计算。如果要同时计算多个数值,应该怎么办?如果要重复计算多次呢?如果要在多个可选项中进行选择应如何做?应该如何获得输入、输出数据?和许多语言一样, C++ 语言也是通过语句来实现这些功能的。

到目前为止,我们已经见过两种语句了:表达式语句和声明语句。表达式语句是以分号结束的一个表达式。例如:

```
a = b;
++b;
```

上面是两个表达式语句的例子。注意, $=$ 是运算符。因此, $a = b$; 是一个以分号结尾的表达式语

句。分号的使用主要是出于技术上的考虑,例如:

```
a = b ++ b; // syntax error: missing semicolon
```

这条语句错误的原因是:如果缺少分号的话,编译器不知道这条语句表示的是 $a = b ++ ; b$; 或者 $a = b ; ++ b ;$ 。这种二义性问题不但存在于编程语言中,也存在于自然语言中。例如,“人吃虎”(man eating tiger)这句话就很令人费解,到底谁吃谁啊?如果加上标点符号就很好理解了:“食人虎”(man-eating tiger)。

计算机是严格按照语句在程序中的书写顺序来执行的,例如:

```
int a = 7;
cout << a << '\n';
```

在这里,变量声明语句及其初始化在输出语句之前执行。

程序中的语句一般都要起作用,我们把不起作用的语句称为无效语句。例如,

```
1+2; // do an addition, but don't use the sum
a*b; // do a multiplication, but don't use the product
```

这类无效语句一般都是逻辑错误造成的,编译器会对这类无效语句给出警告信息。总结一下,表达式语句主要包括赋值语句、I/O 语句和函数调用。

此外,我们还介绍一种其他语句形式:“空语句”。考虑如下代码:

```
if (x == 5);
{ y = 3; }
```

上面的语句看上去是有错误的。事实上,按照程序的原意,它也确实存在语义错误:第一行不应该出现分号结束符。但不幸的是,按照 C++ 语言的语法,这是一个合法的空语句,分号前什么也没有即表示空语句,它什么也不做,很少被使用。但是在上面这个例子中,空语句的存在掩盖了一个语义错误,而且编译器也无法发现这个错误,这样就大大增加了程序员发现错误的难度。

上述代码的执行结果是什么呢?编译器首先会检查 x 的值是否等于 5。如果条件是真,那么将执行接下来的语句(空语句),结果是什么也不做。然后程序执行下一行, y 被赋值为 3(而按照程序的原意,只有当 x 的值等于 5 的时候,才执行赋值操作)。也就是说,这个 if 语句没有起作用:无论 if 语句的结果是什么, y 都被赋值为 3。这是一个新手常犯的错误,而且这种错误很难发现。

4.4.1 选择语句

无论在程序中或者在生活中,我们都会面临各种选择问题。在 C++ 语言中,选择是利用 if 语句或者 switch 语句实现的。

4.4.1.1 if 语句

if 语句是最简单的选择语句,可以在两种可选分支中进行选择。例如,

```
int main()
{
    int a = 0;
    int b = 0;
    cout << "Please enter two integers\n";
    cin >> a >> b;

    if (a < b) // condition
        // 1st alternative (taken if condition is true):
        cout << "max(" << a << ", " << b << ") is " << b << "\n";

    else
        // 2nd alternative (taken if condition is false):
        cout << "max(" << a << ", " << b << ") is " << a << "\n";
}
```

if 语句在两个分支之间进行选择, 如果条件为真, 那么执行第一个分支语句, 否则执行第二个分支语句。大部分编程语言都是这样规定的。事实上, 编程语言的这种规定来自于实际学习和生活中的习惯。例如, 在幼儿园你就应该学习了过马路时要看交通灯: “红灯停, 绿灯行”, 对应的 C++ 程序为:

```
if (traffic_light==green) go();
```

和

```
if (traffic_light==red) wait();
```

虽然 if 语句的基本概念很简单, 但在使用 if 语句时也要仔细。看看下面的程序有什么错误(为了简化, 省略了 #include 语句):

```
// convert from inches to centimeters or centimeters to inches
// a suffix 'i' or 'c' indicates the unit of the input
int main()
{
    const double cm_per_inch = 2.54; // number of centimeters in an inch
    double length = 1;               // length in inches or centimeters
    char unit = 0;
    cout<< "Please enter a length followed by a unit (c or i):\n";
    cin >> length >> unit;

    if (unit == 'i')
        cout << length << "in == " << cm_per_inch*length << "cm\n";
    else
        cout << length << "cm == " << length/cm_per_inch << "in\n";
}
```

实际上, 如果严格按照格式输入数据的话, 这个程序是能够正确执行的: 输入 1i 得到输出 1in == 2.54cm; 输入 2.54c 得到输出 2.54cm == 1in。读者不妨自己试一下。

这个程序的问题在于我们没有测试非法数据输入的情况, 它假定每一次输入都是合法的。程序的条件 unit == 'i' 只是区分了 'i' 和其他的所有情况, 而没有专门针对 'c' 进行判断。

如果用户输入 15f (15 英尺) 会出现什么情况呢? 条件表达式 (unit == 'i') 的值为假。因此, 程序将执行 else 部分 (第二个分支), 即执行厘米到英寸的转换。但是, 这个结果明显不是我们需要的英尺的转换。

除了合法输入情况以外, 程序必须要经过各种非法输入的检验。因为对用户来说, 非法输入是不可避免的。这些非法输入可能是偶然的, 也可能是故意的。但不管用户出于什么目的造成了非法输入的情况, 程序都必须能够检测到。

下面给出了上述代码的改进版本:

```
// convert from inches to centimeters or centimeters to inches
// a suffix 'i' or 'c' indicates the unit of the input
// any other suffix is an error
int main()
{
    const double cm_per_inch = 2.54; // number of centimeters in an inch
    double length = 1;               // length in inches or centimeters
    char unit = ' ';                 // a space is not a unit
    cout<< "Please enter a length followed by a unit (c or i):\n";
    cin >> length >> unit;

    if (unit == 'i')
        cout << length << "in == " << cm_per_inch*length << "cm\n";
    else if (unit == 'c')
        cout << length << "cm == " << length/cm_per_inch << "in\n";
    else
        cout << "Sorry, I don't know a unit called '" << unit << "'\n";
}
```

在这个程序中，依次检验 `unit == 'i'` 和 `unit == 'c'`，如果都不成立，则显示出错信息。看上去好像是我们使用了 `else-if` 语句，但 C++ 语言中没有这种语句。实际上，这里是将两条 `if` 语句组合起来使用的。`if` 语句的一般形式为：

`if (表达式) 语句 else 语句`

关键字 `if` 后面是用括号括起来的表达式，然后是一条语句，`else` 后面是另一个分支语句，其中该分支语句可以是一条 `if` 语句：

`if (表达式) 语句 else if (表达式) 语句 else 语句`

上面所给程序的结构如下：

```
if (unit == 'i')
    ...           // 1st alternative
else if (unit == 'c')
    ...           // 2nd alternative
else
    ...           // 3rd alternative
```

通过这种方式，我们可以写出包含任意分支的复杂语句。但需要注意的是，代码应该尽量简洁，而不是复杂。编写最复杂的代码并不能显示你的智力水平。反之，能够用简洁的代码完成同样的目标才能体现你的能力。

试一试 基于前面给出的示例程序，编写一个能够将日元、欧元和英镑兑换为美元的程序。为了更接近实际情况，你可以从互联网上获得最新的汇率。

4.4.1.2 switch 语句

实际上，示例中的 `unit` 和 `'i'`、`'c'` 的比较是最常见的选择形式：基于数值与多个常量比较的选择。在程序设计中经常会用到这种选择，因此 C++ 语言专门提供了一个语句：`switch` 语句。利用 `switch` 语句可将前面的程序改写为：

```
int main()
{
    const double cm_per_inch = 2.54; // number of centimeters in an inch
    double length = 1;               // length in inches or centimeters
    char unit = 'a';
    cout << "Please enter a length followed by a unit (c or i):\n";
    cin >> length >> unit;
    switch (unit) {
        case 'i':
            cout << length << "in == " << cm_per_inch * length << "cm\n";
            break;
        case 'c':
            cout << length << "cm == " << length / cm_per_inch << "in\n";
            break;
        default:
            cout << "Sorry, I don't know a unit called '" << unit << "'\n";
            break;
    }
}
```

与 `if` 语句相比，`switch` 语句更加清晰易懂，特别是与多个常量进行比较时。关键字 `switch` 后括号中的值与一组常量进行比较，每个常量用一个 `case` 语句标记。如果该值与某一常量相等，将选择执行该 `case` 语句，每个 `case` 语句都以 `break` 结束。如果该值与任何一个 `case` 后的常量都不相等，则选择执行 `default` 语句。虽然 `default` 语句不是必须的，但我们建议你加上，除非你能够完全确定给出的分支已经覆盖了所有的情况。注意，编程能够让你理解世界上没有绝对确定的事情。

4.4.1.3 switch 技术

下面是一些与 switch 语句相关的技术细节：

- 1) switch 语句括号中的值必须是整型、字符型或枚举类型(参见 9.5 节)。特别地,不能使用字符串类型。
- 2) case 语句中的值必须是常量表达式(参见 4.3.1 节),不能使用变量。
- 3) 不能在两个 case 语句中使用相同的数值。
- 4) 允许多个 case 语句使用相同的复合语句。
- 5) 不要忘记在每个 case 语句末尾加上 break。注意,编译器不会给出未加 break 的任何警告信息。

例如：

```
int main()    // you can switch only on integers, etc.
{
    cout << "Do you like fish?\n";
    string s;
    cin >> s;
    switch (s) {    // error: the value must be of integer, char, or enum type
    case "no":
        // ...
        break;
    case "yes":
        // ...
        break;
    }
}
```

如果要对 string 类型的数据进行选择,只能使用 if 语句或者 map(见第 21 章)。

switch 语句能够对一组常量的比较产生优化的代码,特别是当常量数目很多的时候,switch 语句比 if 语句的嵌套使用更加有效。但是,case 语句中的值必须是常量,而且不能重复。例如：

```
int main()    // case labels must be constants
{
    // define alternatives:
    int y = 'y';    // this is going to cause trouble
    const char n = 'n';
    const char m = '?';
    cout << "Do you like fish?\n";
    char a;
    cin >> a;
    switch (a) {
    case n:
        // ...
        break;
    case y:    // error: variable in case label
        // ...
        break;
    case m:
        // ...
        break;
    case 'n':    // error: duplicate case label (n's value is 'n')
        // ...
        break;
    default:
        // ...
        break;
    }
}
```

如果希望采用同样的操作对一组值进行处理,可以为这个操作加上一组标记,而不是重复写同样的操作代码。例如:

```
int main() // you can label a statement with several case labels
{
    cout << "Please enter a digit\n";
    char a;
    cin >> a;

    switch (a) {
        case '0': case '2': case '4': case '6': case '8':
            cout << "is even\n";
            break;
        case '1': case '3': case '5': case '7': case '9':
            cout << "is odd\n";
            break;
        default:
            cout << "is not a digit\n";
            break;
    }
}
```

使用 switch 语句时,常犯错误是忘记为 case 语句添加 break。例如:

```
int main() // example of bad code (a break is missing)
{
    const double cm_per_inch = 2.54; // number of centimeters in an inch
    double length = 1; // length in inches or centimeters
    char unit = 'a';
    cout << "Please enter a length followed by a unit (c or i):\n";
    cin >> length >> unit;

    switch (unit) {
        case 'i':
            cout << length << "in == " << cm_per_inch * length << "cm\n";
        case 'c':
            cout << length << "cm == " << length / cm_per_inch << "in\n";
    }
}
```

不幸的是,对于上面这个例子,编译器是不会报错的。当执行完 case 'i' 的代码后,程序接着执行 case 'c' 的代码。如果输入 2i 的话,程序将输出

```
2in == 5.08cm
2cm == 0.787402in
```

这个问题需要特别注意!

试一试 用 switch 语句重写 4.4.1.1 节的“试一试”中给出的汇率转换程序,并且增加人民币和克朗的转换功能。哪一个版本的程序更容易编写、理解和修改呢?为什么?

4.4.2 循环语句

现实生活中,我们经常会遇到一些重复性的工作。为此,编程语言也提供了相应的语言工具,称为循环(repetition)。在对一系列数据进行同样处理的时候,也称它为迭代(iteration)。

4.4.2.1 While 语句

在世界上第一台能存储程序的计算机(名为 EDSAC)上运行的第一个程序就是一个循环语句程序,它是由英国剑桥大学计算机实验室的 David Wheeler 在 1949 年 5 月 6 日编写的,其目的是计算并打印下面这个简单的平方表:

```

0    0
1    1
2    4
3    9
4    16
...
98   9604
99   9801

```

平方表的每一行是一个数，后面跟着一个制表符('\t')，然后是该数的平方。该程序的 C++ 版本如下：

```

// calculate and print a table of squares 0-99
int main()
{
    int i = 0;    // start from 0
    while (i < 100) {
        cout << i << '\t' << square(i) << '\n';
        ++i;    // increment i (that is, i becomes i+1)
    }
}

```

程序中的 `square(i)` 表示 i 的平方，其含义和用法将在后续章节中解释(参见 4.5 节)。

实际上，这个程序并不是一个真正的 C++ 程序。它的程序逻辑如下：

- 从 0 开始计数。
- 检查计数是否达到 100，如果是的话，程序结束。
- 否则，打印这个数和它的平方，中间用制表符('\t')隔开。计数加 1，重复上述操作。

显然，完成上述目标，我们需要

- 一种实现语句重复执行的方法(循环)。
- 一个记录循环次数的变量(循环变量或控制变量)，在上面的例子中是整型变量 i 。
- 循环变量的初始化，在示例中是 0。
- 循环结束条件，在示例中是循环 100 次。
- 每次循环中完成的操作(循环体)。

这里使用 `while` 语句实现这个功能。在 `while` 语句中，关键字 `while` 之后是循环条件，然后是循环体。

```

while (i < 100)    // the loop condition testing the loop variable i
{
    cout << i << '\t' << square(i) << '\n';
    ++i;    // increment the loop variable i
}

```

循环体是一个程序块，其任务是输出平方表的一行，并将循环控制变量 i 的值加 1。每次循环开始都要检查循环条件 $i < 100$ 是否成立，若成立则执行循环体；如果不成立，即 i 的值达到 100 的时候，则结束 `while` 语句，执行后续的程序代码。在上面的示例中，`while` 语句是程序的最后一条语句。因此，`while` 语句结束后程序也随之结束。

`while` 语句的循环控制变量必须在 `while` 语句之前定义和初始化，否则编译器将返回一个错误。如果定义了循环控制变量而没有初始化，大部分编译器会返回一个警告信息“本地变量 i 没有赋初值”，但不会作为编译错误来处理。请注意，编译器给出的变量未初始化的信息大都是对的，未初始化的变量会导致很多错误。在上面的示例中，应该加上下面的语句。

```

int i = 0;    // start from 0

```

编写循环语句很简单，但让循环语句能够准确反映实际问题却很困难。其中，主要难点是如何让循环正确地开始和结束，这里的关键是循环条件的设置和所有变量的初始化。

试一试 字符'b'可以通过 `char('a' + 1)` 得到, 字符'c'可以通过 `char('a' + 2)` 得到。

用一个循环语句来实现字母表及其相应 ASCII 码值的输出:

```
a    97
b    98
...
z    122
```

4.4.2.2 程序块

注意下面程序中 `while` 语句的循环体是如何定义的:

```
while (i<100) {
    cout << i << '\t' << square(i) << '\n';
    ++i;    // increment i (that is, i becomes i+1)
}
```

我们把用 `{` 和 `}` 括起来的语句序列称为程序块 (block) 或复合语句 (compound statement)。程序块是一种特殊语句, 不包括任何具体语句的程序块也是有用的, 它表示什么也不做。例如:

```
if (a<=b) {    // do nothing
}
else {        // swap a and b
    int t = a;
    a = b;
    b = t;
}
```

4.4.2.3 for 语句

像大多数编程语言一样, C++ 为针对一组数据的迭代操作设置了专门的语句。`for` 语句与 `while` 语句类似, 只是 `for` 语句要求将循环控制变量集中放在开头, 以便于阅读和理解。使用 `for` 语句的第一个示例如下:

```
// calculate and print a table of squares 0-99
int main()
{
    for (int i = 0; i<100; ++i)
        cout << i << '\t' << square(i) << '\n';
}
```

该程序的含义是“从 `i` 等于 0 开始执行循环体, 每执行一次循环体, `i` 的值加 1, 直到 100 为止”。`for` 语句可以用等价的 `while` 语句来替换, 例如:

```
for (int i = 0; i<100; ++i)
    cout << i << '\t' << square(i) << '\n';
```

等价于

```
{
    int i = 0;           // the for-statement initializer
    while (i<100) {      // the for-statement condition
        cout << i << '\t' << square(i) << '\n';    // the for-statement body
        ++i;             // the for-statement increment
    }
}
```

有的初学者喜欢使用 `while` 语句, 有的则喜欢使用 `for` 语句。与 `while` 语句相比, `for` 语句的代码更容易理解和维护。这是因为 `for` 语句将循环相关的初始化、循环条件和增量操作集中放在一起, 而 `while` 语句则不是这样。

注意, 不要在 `for` 语句的循环体内修改循环控制变量的值。这种操作虽然没有语法错误, 但是它违背了读者对于循环的普遍理解和认识。考虑下面这个例子:

```
int main()
{
    for (int i = 0; i<100; ++i) {    // for i in the [0:100) range
        cout << i << '\t' << square(i) << '\n';
        ++i;    // what's going on here? It smells like an error!
    }
}
```

乍看起来，每个人都认为上面的循环会执行 100 次，但实际上不到 100 次。循环体中的 `++i` 语句使得 `i` 的值每执行一次循环体就加 2，所以只执行了 50 次。这个程序虽然没有语法错误，但是我们看到这样的程序仍然认为程序有错，错误原因也许就是把 `while` 语句转换为 `for` 语句时的粗心大意造成的。如果我们希望循环控制变量每次加 2，可以像下面这么写：

```
// calculate and print a table of squares of even numbers in the [0:100) range
int main()
{
    for (int i = 0; i<100; i+=2)
        cout << i << '\t' << square(i) << '\n';
}
```

此外，还需要注意程序的简洁性，后面将介绍一些典型的示例。

试一试 使用 `for` 语句重写 4.4.2.1 节的字符输出程序，并修改你的程序，使其还可以输出所有大写字母和数字。

4.5 函数

在上面的程序中 `square(i)` 是什么呢？它是一个函数调用。准确地说，它使用参数 `i` 调用 `square` 函数。函数 (function) 是一个命名的语句序列，能够返回计算结果 (称为返回值)。C++ 的标准库提供了许多有用的函数，例如在 3.4 节中用到的求平方根函数 `sqrt()`，但我们在程序中还需要写很多函数。`square` 函数的一种可行定义如下：

```
int square(int x)    // return the square of x
{
    return x*x;
}
```

第一行说明这是一个名为 `square` 的函数 (由括号可知)，它有一个 `int` 型参数 (名为 `x`)，返回值也是 `int` 型 (函数定义中的第一个关键字)。这个函数的使用如下：

```
int main()
{
    cout << square(2) << '\n';    // print 4
    cout << square(10) << '\n';    // print 100
}
```

对于函数的返回结果，我们可以使用也可以不使用。(但如果我们不需要函数返回结果的话，为什么还要调用它呢?) 但是，我们必须严格按照函数的定义给它传递参数，例如：

```
square(2);                // probably a mistake: unused return value
int v1 = square();         // error: argument missing
int v2 = square;           // error: parentheses missing
int v3 = square(1,2);      // error: too many arguments
int v4 = square("two");    // error: wrong type of argument — int expected
```

很多编译器都会警告未使用的函数返回值，并给出上面示例中的错误信息。你可能会认为计算机很“聪明”，它应该能够理解“two”表示整数 2。但实际上，C++ 编译器并不像你想象的那样。编译器的工作是检查你的代码是否符合 C++ 语言规范，并严格按照你的程序要求去执行。如果让编译器去猜测你的真实意图的话，那么它很可能会猜错，从而导致你或者你的程序用户陷入麻

烦。你将会发现如果没有编译器的猜测等“帮助”，很难预测程序的运行结果。

函数体(function body)是实现某种具体功能的程序块(参见4.4.2.2节)。

```
{
    return x*x; // return the square of x
}
```

函数 square 的实现比较简单：计算参数的平方，并将计算结果作为函数返回值。显然，用 C++ 语言描述比用自然语言(英语、汉语等)描述更简洁。这一点在很多情况下都适用，毕竟，程序语言的目的就是用一种更简洁、准确的方式来描述我们的思想。

函数定义(function definition)的语法描述如下：

类型 函数名(参数表)函数体

其中，类型是函数的返回值类型，函数名是函数的标记，括号内是参数表，函数体是实现函数功能的语句。参数表(parameter list)的每一个元素称为一个参数(parameter)或形式参数(formal argument)，参数表可以为空。如果不需要函数返回任何结果，返回值类型可以设置为 void。例如：

```
void write_sorry() // take no argument; return no value
{
    cout << "Sorry\n";
}
```

函数语法的相关细节可以参考本书第8章的内容。

4.5.1 使用函数的原因

当需要将一部分计算任务独立实现的时候，可以将其定义为一个函数，因为这样可以：

- 实现计算逻辑的分离。
- 使代码更清晰(通过使用函数名)。
- 利用函数，使得同样的代码在程序中可以多次使用。
- 减少程序调试的工作量。

在本书的后续内容中，我们将看到很多解释上述原因的示例，并且我们还会再次谈及某个原因。注意，实际的应用程序可能会用到成百上千个函数，某些程序甚至会用到上百万个函数。显然，如果这些函数不能被清楚地划分和命名的话，任何人都不能编写和理解这种包含大量函数的程序。而且，你会发现很多函数被重复使用，很快这将导致你厌倦于这种重复性的劳动。例如，编写处理 $x * x$ 、 $7 * 7$ 和 $(x + 7) * (x + 7)$ 等的程序可能会令你高兴。但是，对完成同样功能的函数 $\text{square}(x)$ 、 $\text{square}(7)$ 和 $\text{square}(x + 7)$ 却可能会让你厌倦。这是因为，求平方是非常容易实现的，但对于复杂函数来说，情况大不相同：对于求平方根函数(C++中的 sqrt)来说，程序员更喜欢使用 $\text{sqrt}(x)$ 、 $\text{sqrt}(7)$ 和 $\text{sqrt}(x + 7)$ ，而不是每一次都重复实现相同的求平方根的代码(这些代码很长、很复杂)。实际上，大多数情况下，你根本不需要了解求解平方根函数的实现细节，而只需要知道 $\text{sqrt}(x)$ 将返回 x 的平方根就可以了。

在8.5节中，我们将详细介绍相关的函数编写技巧。下面我们将给出另外一个示例。

如果我们想让主函数中的循环更简洁，可将程序改写为：

```
void print_square(int v)
{
    cout << v << '\t' << v*v << '\n';
}

int main()
{
    for (int i = 0; i < 100; ++i) print_square(i);
}
```


但我们为什么不用 `print_square()` 版本的程序呢？因为这个版本的程序实际上并不比 `square()` 版本的程序更简洁。因为：

- `print_square()` 是一个比较特殊的函数，以后很难再次用到它，而 `square()` 可以多次重复使用。
- `square()` 几乎不需要任何额外的说明文档，而 `print_square()` 需要相关文档说明函数的功能和使用方法等。

根本的原因是 `print_square()` 需要执行两个独立的逻辑操作：

- 输出结果。
- 计算平方值。

如果每个函数只完成单一的逻辑操作，那么程序将更易于编写和理解。因此，使用 `square()` 是一个更好的选择。

最后，为什么我们要用 `square()` 而不是第一个版本程序中的 `i*i` 呢？使用函数的目的之一是用函数把一些复杂的运算分离出来。对程序的 1949 年版本来说，硬件没有提供对乘法操作的直接支持，因此，1949 年版本程序中的乘法是一个类似笔算的复杂计算过程。而且，1949 年版本程序的作者 (David Wheeler) 也是现代计算中的函数 (称为子程序) 的发明者，这也是使用 `square()` 的原因。

试一试 不用乘法操作实现 `square()` 的功能，即利用重复加法操作实现 `x*x` (设置一个初值为 0 的变量，把 `x` 的值加到该变量上 `x` 次)。然后，利用这一 `square()` 运行前面的示例。

4.5.2 函数声明

你是否注意到，函数调用所需的所有信息都已经包括在函数定义的第一行？例如：

```
int square(int x)
```

根据这些信息，可以写出如下语句：

```
int x = square(44);
```

我们不需要知道函数体是如何实现的。在编写程序的时候，我们一般不需要知道函数体的实现细节。为什么我们要知道标准库函数 `sqrt()` 是如何实现的呢？我们知道它能够计算参数的平方根。为什么我们要阅读 `square()` 函数的代码呢？虽然我们可能会好奇它的具体实现，但大多数情况下，我们仅仅关心如何调用函数就可以了。幸运的是，C++ 提供了一种与函数定义分离的方法来显示函数的信息，称为函数声明 (function declaration)。

```
int square(int);           // declaration of square
double sqrt(double);       // declaration of sqrt
```

注意，函数声明以分号结束，分号替代了函数定义中的函数体部分：

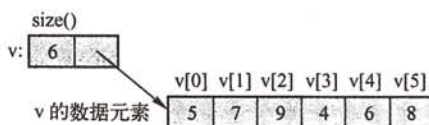
```
int square(int x)  // definition of square
{
    return x*x;
}
```

如果你想使用某个函数，可以在代码中声明或者通过 `#include` 包含该函数的函数声明，而函数的定义可以在程序的其他部分，我们将在 8.3 节和 8.7 节中讨论具体的实现细节。函数声明和函数定义的分离对于大型程序是非常必要的，我们可以用函数声明来保证代码的简洁，从而保证在同一时刻将注意力集中在程序的某一局部区域上 (参见 4.2 节)。

4.6 向量

在编写程序之前，我们首先要准备好相关的数据。例如，我们可能需要准备一组电话号码，一个球队的队员名单，一个课程表，最近一年读的书的列表，下载歌曲的分类表，汽车付款的可选途径，下周每一天的天气预测情况，同一款相机在不同网上商店的价格对比表等。程序可能用到的各种数据形式数不胜数，并存储在于程序的所有代码中。我们将从数据的各种存储形式开始介绍（其他数据存储形式介绍参考本书第20章和第21章）。最简单、最常用的数据存储形式是向量。

向量是一组可以通过索引来访问的顺序存储的数据元素。例如，右图是一个名为 *v* 的向量。其中，第一个数据元素的索引号是 0，第二个是 1，依此类推。我们可以用向量名和索引号的组合来表示一个具体的数据元素，例



如 *v*[0] 是 5，*v*[1] 是 7，依此类推。向量的索引号总是从 0 开始，每次加 1。看上去有些熟悉，实际上向量是 C++ 标准库函数中一个历史久远的著名库函数实现的简化版本。图中特别强调了向量 *v* “知道自己的大小”，即向量不仅存储数据元素，也存储元素的个数。

向量可以用如下形式表示：

```
vector<int> v(6); // vector of 6 ints
v[0] = 5;
v[1] = 7;
v[2] = 9;
v[3] = 4;
v[4] = 6;
v[5] = 8;
```

可以看出，定义一个向量需要确定向量的数据类型和元素个数。数据类型在紧跟向量名的 < > 内定义（如 < int >），向量包含的元素个数在后面的圆括号内定义（如 (6)）。下面是另一个示例：

```
vector<string> philosopher(4); // vector of 4 strings
philosopher[0] = "Kant";
philosopher[1] = "Plato";
philosopher[2] = "Hume";
philosopher[3] = "Kierkegaard";
```

显然，一个向量只能存储与其数据类型相同的数据：

```
philosopher[2] = 99; // error: trying to assign an int to a string
v[2] = "Hume"; // error: trying to assign a string to an int
```

当一个给定大小的向量被定义后，根据数据类型的不同，它的每一个数据元素将被赋予不同的初值，例如：

```
vector<int> v(6); // vector of 6 ints initialized to 0
vector<string> philosopher(4); // vector of 4 strings initialized to ""
```

当然，也可以为向量赋予你希望的初值。例如：

```
vector<double> vd(1000, -1.2); // vector of 1000 doubles initialized to -1.2
```

注意，不能引用一个不存在的向量元素。例如：

```
vd[20000] = 4.7; // run-time error
```

我们将在第 5 章详细讨论关于运行错误的细节。

4.6.1 向量空间增长

我们使用向量的时候，一般是从一个空向量开始，根据需要逐步填充数据。这里的关键操作是 *push_back()*，它将一个新元素添加到向量中，该元素成为向量的最后一个元素。例如：

```
vector<double> v;    // start off empty; that is, v has no elements
```

v: 

```
v.push_back(2.7);    // add an element with the value 2.7 at end ("the back") of v
                     // v now has one element and v[0]==2.7
```

v: 

```
v.push_back(5.6);    // add an element with the value 5.6 at end of v
                     // v now has two elements and v[1]==5.6
```

v: 

```
v.push_back(7.9);    // add an element with the value 7.9 at end of v
                     // v now has three elements and v[2]==7.9
```

v: 

注意 `push_back()` 的调用方法, 这是一个成员函数调用 (member function call)。`push_back()` 是向量的一个成员函数, 因此它的调用必须采用符号“.”:

成员函数调用:

对象名. 成员函数名(参数表)

向量的大小可以通过调用成员函数 `size()` 来获得。初始时 `v.size()` 的值是 0, 三次调用 `push_back()` 之后, `v.size()` 的值变为 3。在遍历向量元素的时候, 向量的大小可以作为循环控制变量。例如:

```
for(int i=0; i<v.size(); ++i)
    cout << "v[" << i << "]==" << v[i] << '\n';
```

给定上述 `v` 和 `push_back()` 的定义之后, `for` 循环将输出:

```
v[0]==2.7
v[1]==5.6
v[2]==7.9
```

如果你以前写过程序, 你会注意到向量非常类似于 C 语言中的数组。但在向量中, 你不需要事先指定向量的大小, 而且你可以往向量中添加任意多个元素。后面还将发现 C++ 语言的向量有更多有价值的特性。

4.6.2 一个数值计算的例子

让我们来看一个更实际的例子。我们经常会遇到把一系列数据读入程序来处理的情况, 这些数据处理操作包括: 根据数据显示图形, 计算平均值和中值, 找出最大元素、排序、数据融合、搜索、与其他数据的比较等。对于数据的处理操作是没有任何限制的, 但在做各种数据处理前, 必须先把数据读入内存中。下面是一种把未知大小(可能很大)的数据读入计算机的基本方法。不失一般性, 我们选择读入表示温度的一系列浮点数:

```
// read some temperatures into a vector
int main()
{
    vector<double> temps;           // temperatures
    double temp;
    while (cin>>temp)              // read
        temps.push_back(temp);    // put into vector
    // ... do something ...
}
```

我们得到什么了呢? 首先, 我们声明了一个用于存储数据的向量, 然后通过一个临时变量将数据输入这个向量:

```
vector<double> temps;           // temperatures
double temp;
```

这两条语句说明了我们希望使用的数据类型，此处为 double 类型。

接下来是实际的读循环：

```
while (cin>>temp)           // read
    temps.push_back(temp); // put into vector
```

语句 `cin >> temp` 读入一个 double 类型的数据，然后将这个数据置入向量中（放在最后）。这里用到的每个操作在前面的示例中几乎都出现过，只是这里使用输入语句 `cin >> temp` 作为 while 循环的条件。如果正确输入数据，`cin >> temp` 返回 true，否则返回 false。因此，while 循环将读入我们输入的所有 double 类型的数据，直至读到一个其他类型的数据为止。例如，如果输入

```
1.2 3.4 5.6 7.8 9.0]
```

那么向量 temps 将顺序输入 5 个元素 1.2、3.4、5.6、7.8 和 9.0（例如 `temps[0] == 1.2`）。接着，我们用一个字符 ']' 结束输入，实际上，任何一个非 double 类型的数据都可以作为输入的结束标志。在 10.6 节中，我们还将讨论如何终止输入和处理输入错误。

一旦数据存入向量中，就可以方便地对其进行处理。下面的例子用于计算温度的平均值和中值：

```
// compute mean and median temperatures
int main()
{
    vector<double> temps;           // temperatures
    double temp;
    while (cin>>temp)               // read
        temps.push_back(temp);     // put into vector

    // compute mean temperature:
    double sum = 0;
    for (int i = 0; i < temps.size(); ++i) sum += temps[i];
    cout << "Average temperature: " << sum/temps.size() << endl;

    // compute median temperature:
    sort(temps.begin(), temps.end()); // sort temps
                                     // "from the beginning to the end"
    cout << "Median temperature: " << temps[temps.size()/2] << endl;
}
```

我们可以用一种简单的方法计算向量平均值：把所有元素累加到变量 sum 中，然后除以元素的个数（即 `temps.size()`）：

```
// compute average temperature:
double sum = 0;
for (int i = 0; i < temps.size(); ++i) sum += temps[i];
cout << "Average temperature: " << sum/temps.size() << endl;
```

注意 += 运算符的使用方法。

如果要计算向量的中值，必须先对向量进行排序（中值是指序列中大于一半元素而小于另一半元素的那个元素）。我们使用了标准库排序算法 `sort()`：

```
// compute median temperature:
sort(temps.begin(), temps.end()); // sort "from the beginning to the end"
cout << "Median temperature: " << temps[temps.size()/2] << endl;
```

`sort()` 有两个参数：待排序元素序列的开始和结束。在第 20 章中我们将详细解释标准库函数的算法实现细节。幸运的是，向量能够“知道”开始和结束的具体位置：`temps.begin()` 和 `temps.end()` 完全能够做到这一点。与 `size()` 一样，`begin()` 和 `end()` 都是向量的成员函数。我们可以用“向量名.”的方式调用它们。在排序所有的温度数据后，求中值就很简单了：中值就是下标为 `temps.size()/2` 的那个元素。再深入思考一下，你会发现我们得到的结果有可能不是按照中值定义得到的结果（如果

你这么想了，恭喜你，你已经开始像一名程序员一样思考了)。本章的习题2将解决这个小问题。

4.6.3 一个文本处理的例子

本节我们不再用温度的例子，因为我们对温度数据并不是特别有兴趣。对于气象学家、农学家、海洋学家等，温度以及基于温度的各类数据是非常重要的。但从程序员的角度看，我们感兴趣的是数据组织的一般形式：可以用于各种应用的向量以及对向量的各种操作。总之，不管你对什么内容感兴趣，只要进行数据分析就必须使用向量(或者类似数据结构，具体内容参考第21章)。下面的例子说明如何建立一个简单的字典：

```
// simple dictionary: list of sorted words
int main()
{
    vector<string> words;
    string temp;
    while (cin>>temp)           // read whitespace-separated words
        words.push_back(temp);  // put into vector
    cout << "Number of words: " << words.size() << endl;

    sort(words.begin(),words.end()); // sort "from beginning to end"

    for (int i = 0; i < words.size(); ++i)
        if (i==0 || words[i-1]!=words[i]) // is this a new word?
            cout << words[i] << "\n";
}
```

程序会按照字典序输出向程序输入的单词，同时消除重复的单词。例如，输入

a man a plan panama

程序将输出

a
man
panama
plan

那我们应该如何停止输入呢？或者说，我们应该如何终止这个输入循环呢？

```
while (cin>>temp)           // read
    words.push_back(temp);  // put into vector
```

在做数值的读入操作时(参见4.6.2节)，我们可以通过输入非数值字符来结束输入。但在这个程序中，这种方法是不行的。因为所有的输入字符都被存入一个字符串。幸运的是，我们可以利用一些特殊字符作为终止符。在3.5.1节中介绍了Ctrl+Z可以终止Windows窗口中的一个输入流，Ctrl+D可以终止一个UNIX窗口的输入流。

大多数这类程序与前面的温度程序是非常类似的。事实上，我们在写“简单字典”的程序时，很多代码是从“温度程序”中复制过来的，这里只是添加了对单词重复性的判断：

```
if (i==0 || words[i-1]!=words[i]) // is this a new word?
```

如果删除这条语句，则输出结果为：

a
a
man
panama
plan

我们不喜欢重复，上面的语句消除了重复性的单词，它是怎样做到的呢？该语句检查前一个已输出的单词是否与下一个即将输出的单词相同(words[i-1] != words[i])：如果不同就输出这个单词，否则不输出。显然，在打印第一个单词的时候，不存在前一个单词。因此，还需要判断是否

是第一个单词($i == 0$)。把上面两种判断条件用逻辑或($||$)组合起来可得:

```
if (i==0 || words[i-1] != words[i])    // is this a new word?
```

注意,在比较字符串时可以使用 $!=$ (不等于)、 $==$ (等于)、 $<$ (小于)、 $<=$ (小于等于)、 $>$ (大于)和 $>=$ (大于等于)等。这些关系运算符依据字典序进行判断,因此 "Ape" 在 "Apple" 和 "Chimpanzee" 之前。

试一试 编写一个程序,当遇到你不喜欢的单词时会蜂鸣提醒。具体实现思路如下:用 `cin` 输入单词并用 `cout` 输出。如果一个单词属于你预先定义的不喜欢单词的集合,那么程序输出 BLEEP 而不是这个单词本身。可以用如下方法定义不喜欢单词:

```
string disliked = "Broccoli";
```

实现程序功能后,可以尝试增加更多单词。

4.7 语言特性

温度和字典程序用到了本章介绍的大部分语言特性:循环(`for` 语句和 `while` 语句)、选择语句(`if` 语句)、简单的算术运算($++$ 和 $+=$ 操作)、比较和逻辑运算($==$, $!=$ 和 $||$)、变量和函数(例如 `main()`, `sort()` 和 `size()` 等)。此外,还用到了标准库函数,例如向量(一种数据元素的容器)、`cout` (标准输出流)和 `sort()` (一种排序算法)。

仔细回想一下,你会发现我们已经用到了很多语言特性。每一种语言特性都表示了一种基本思想,将许多种语言特性结合起来,我们就能写出有用的程序了。记住,计算机不是一种只能完成固定功能的设备,它可以完成我们能想象到的任何计算任务。而且,通过计算机与其他设备的结合,理论上我们可以用它来完成任何任务。

简单练习

请逐步完成下列练习,不要贪快而跳过这些简单练习。请至少使用三组不同的数据来测试这些程序,欢迎使用更多的测试数据。

1. 编写一个使用 `while` 循环语句的程序,每次循环输入两个 `int` 数据并输出它们,当输入 `'l'` 后程序结束。
2. 修改程序,输出 "the smaller value is:" 后接着输出较小的整数,输出 "the larger value is:" 后接着输出较大的整数。
3. 改进程序:当两个整数相等时,输出 "the numbers are equal"。
4. 修改程序:输入的数据改为 `double` 型而不是 `int` 型。
5. 修改程序:如果两个数的差小于 $1.0/100$ 的话,输出 "the numbers are almost equal",然后依次输出较大和较小的数。
6. 修改循环体程序:每次循环只输入一个 `double` 型数据,并用两个变量记录到目前为止的最小数和最大数。每次循环输出当前输入的数据,如果这个数据是到目前为止最小或最大的数,在其后分别输出 "the smallest so far" 和 "the largest so far"。
7. 为每个 `double` 型数据增加单位:例如数据可以是 10cm、2.5in、5ft 或 3.33m。程序可以接受四种计量单位:cm、m、in 和 ft,假定转换系数是 $1\text{m} == 100\text{cm}$ 、 $1\text{in} == 2.54\text{cm}$ 、 $1\text{ft} == 12\text{in}$ 。注意,将单位读入一个字符串。
8. 改进程序使之拒绝没有单位或单位非法的数据,例如 y、yard、meter、km 和 gallons 等。
9. 除了记录到目前为止最大和最小的数据外,还记录到目前为止的数据累加和以及数据个数。当遇到输入 `'l'` 后,程序输出最小、最大、累加和以及数据个数。注意,因为计算累加和必须使用同一计量单位,所以需要事先决定使用哪个计量单位,例如米。
10. 将上述所有的数据(转换为公尺)存入一个向量变量,然后输出这些数据。
11. 在输出向量中的数据前,按照升序将这些数据排序。

思考题

1. 什么是计算?
2. 什么是计算的输入和输出? 举例说明。
3. 当表示计算的时候, 程序员需要谨记哪三项要求?
4. 什么是表达式?
5. 在本章内容中, 表达式和语句有什么区别?
6. 什么是左值? 列出要求用左值的运算符。为什么这些运算符需要使用左值?
7. 什么是常量表达式?
8. 什么是字面常量?
9. 什么是符号常量, 我们应该如何使用它?
10. 什么是魔数? 举例说明。
11. 哪些运算符既可以用于整型也可以用于浮点型?
12. 哪些运算符只能用于整型而不能用于浮点型?
13. 哪些运算符可以用于字符串?
14. 什么情况下程序员更喜欢用 switch 语句而不是 if 语句?
15. switch 语句常见的问题有哪些?
16. for 循环语句的循环控制中每一部分的功能是什么? 它们的执行顺序是怎样的?
17. 什么情况下应该使用 for 循环? 什么情况下应该使用 while 循环?
18. 如何输出一个字符型数据的 ASCII 值?
19. 请说明在函数定义中 `char foo(int x)` 的含义是什么?
20. 什么情况下你将在程序中定义一个单独的函数? 列出原因。
21. 哪些操作可以用于整型数据而不能用于字符串?
22. 哪些操作可以用于字符串而不能用于整型数据?
23. 向量中第三个元素的索引号是多少?
24. 如何用 for 循环来打印输出向量的所有数据元素?
25. 语句 `vector <char> alphabet(26);` 的含义是什么?
26. 描述向量中的 `push_back()` 的含义。
27. `vector` 的成员函数 `begin()`、`end()` 和 `size()` 的功能是什么?
28. 为什么向量被如此广泛地使用?
29. 如何将一个向量中的数据排序?

术语

抽象	for 语句	<code>push_back()</code>	<code>begin()</code>	函数	重复
计算	if 语句	右值	条件语句	增量	选择
声明	输入	<code>size()</code>	定义	迭代	<code>sort()</code>
分治	循环	语句	else	左值	switch 语句
<code>end()</code>	成员函数	vector	表达式	输出	while 语句

习题

1. 如果你还没有完成本章中的“试一试”, 请先完成相关练习。
2. 定义一个序列的中值恰好是序列的一半元素在它之前而另一半元素在它之后的数值, 修改 4.6.2 节的程序使其总是能够输出中值。提示: 中值并不一定是序列中的元素。
3. 输入一组 `double` 型数据到一个向量中, 假设这些数据是沿着某一条路径的相邻城市间的距离。要求: 计算并输出全部距离(所有距离的总和); 搜索并输出相邻两个城市间的最小和最大距离; 搜索并输出两个相邻城市间的平均距离。

4. 编写一个猜数游戏程序。用户给出一个 1 到 100 之间的整数, 程序通过提问来猜测用户所想的数是什么(例如, “你的数小于 50 吗?”), 程序应该能够用不超过 7 个问题来确定这个数。提示: 使用 $<$ 和 $<=$ 运算符以及 if-else 语句编写程序。
5. 实现一个简单的计算器程序。计算器应该能够对两个输入数据实现基本的数学操作: 加、减、乘、除和取模(余数)。程序应该提示用户输入三个参数: 两个 double 型数据和一个表示操作的字符。例如, 如果输入参数 35.6、24.1 和 '+', 程序将输出“35.6 与 24.1 的和等于 59.7”。在第 6 章我们将介绍一个更复杂的计算器程序。
6. 定义一个能够存储 10 个字符串的向量, 分别是“zero”、“one”、...“nine”。编写一个能够实现数字与其对应拼写进行转换的程序。例如, 当输入 7 的时候输出 seven。同时, 该程序还能够实现拼写形式到数字形式的转换, 例如, 当输入 seven 的时候输出 7。
7. 修改习题 5 的“迷你计算器”程序, 使程序不但能够接受数字形式的数据, 也能够接受拼写形式的数据。
8. 有一个古老的故事, 讲述的是一个皇帝为了感谢国际象棋的发明人, 答应这个发明人可以提出自己的赏赐要求。发明人提出的要求是: 在棋盘的第一个格子里放 1 粒米, 在第二个格子里放 2 粒米, 第三个格子里放 4 粒米, 依此类推。每次都加倍直到放满棋盘的所有 64 个格子为止。这个要求听起来很谦虚, 但实际上全国所有的米都不够支付这个赏赐。编写程序来计算一下发明家要获得至少 1000 粒米需要多少个棋盘格子? 至少 1000000 粒米呢? 至少 1000000000 粒米呢? 当然, 你需要设计一个循环, 也许还需要一个整型变量记录当前所处的格子, 一个整型变量记录当前格子的米粒数, 一个整型变量记录以前所有格子的米粒数。建议你在每次循环中都输出所有的变量值, 并观察一下会发生什么情况。
9. 尝试计算习题 8 中发明家要求的米粒的总量是多少? 你会发现这个数字是如此大, 以至于 int 或 double 都无法保存。注意观察当数值太大导致 int 或 double 无法保存时会发生什么。如果使用 int 的话, 你可以准确计算出米数总量的最大棋盘的格子数目是多少? 使用 double 呢?
10. 编写一个“石头、剪刀、布”的游戏程序。如果你不熟悉这个游戏, 可以先调查一下。对于程序员来说, 调查是日常工作的一部分。用 switch 语句解决这个习题。程序将随机给出下一个操作(即石头、剪刀或布是随机出现的)。目前, 真正的随机性是很难实现的, 因此在程序中可以用存有一个数据序列的向量来处理, 其中向量的每个数据元素表示程序的下一个操作。根据这个向量中的数据元素, 程序的每次运行将执行相同的动作, 因此你需要用户输入某些值。尝试做一些变化, 使得用户很难猜出程序的下一个动作是什么。
11. 编写程序找出 1 到 100 之间的所有素数。一种可行的方法是使用一个函数来判断一个数是否是素数(即判断一个数是否能够被小于它的素数整除), 可以将素数存储在一个 vector 类型的素数表中(例如, 如果这个 vector 变量名为 primes、primes[0] == 2、primes[1] == 3、primes[2] == 5 等)。然后用一个 1 到 100 的循环逐个判断每个数是否是素数, 并将其中的素数存储在一个 vector 中。然后, 编写另一个循环来显示所有的素数。你可以比较一下你找到的素数和素数表 primes 的结果。其中, 2 是第一个素数。
12. 修改上面的习题, 要求程序有一个输入值 max, 并找出从 1 到 max 的所有素数。
13. 使用名为“埃拉托斯特尼筛法”(Sieve of Eratosthenes)的经典方法, 编写程序找出 1 到 100 之间的所有素数。如果不了解这个方法, 你可以通过互联网搜索相关资料。
14. 修改上面的习题, 要求程序有一个输入值 max, 并找出从 1 到 max 的所有素数。
15. 编写程序, 要求: 有一个输入值 n, 输出结果是前 n 个素数。
16. 编写程序, 找出一组输入数据中最大和最小的数据。在一组数据中出现次数最多的数称为 mode。要求: 输入一组正整数, 程序能够找出该组数据的 mode。
17. 编写程序, 要求: 输入一组字符串, 找出该组字符串中最大、最小和 mode 字符串。
18. 编写解一元二次方程的程序。一元二次方程的一般形式为

$$ax^2 + bx + c = 0$$

如果你不知道如何解一元二次方程, 可以先调查一下。记住, 在一个程序员教会计算机如何解决问题前, 程序员必须先清楚如何解决它。程序的输入数据 a、b 和 c 为 double 型; 一元二次方程有两个解, 因

此程序的输出包括两个解 x_1 和 x_2 。

19. 编写程序，其输入是一组名字和数值对。例如，Joe 17 和 Barbara 22 等。对于每一个名字 - 数值对，名字存入名为 `names` 的向量中，数值存入名为 `scores` 的向量对应位置中（例如，如果 `names[7] = "Joe"`，那么 `scores[7] = 17`）。当输入 No more 时，终止输入（这里 more 将导致读入一个整型数据的失败）。注意，要检查名字的唯一性，相同的名字将导致程序中断并输出一个错误信息。最后，按照每行一个名字 - 数值对的形式输出所有数据。
20. 修改习题 19 的程序，当你输入一个名字后，程序将输出相应的成绩或者输出“name not found”。
21. 修改习题 19 的程序，当你输入一个整数后，程序将输出所有对应的名字或者“score not found”。

附言

从哲学的观点看，用计算机能做的所有事情，你现在都可以去做了，剩下的就是一些具体细节了。由于你是一个编程的初学者，我们要郑重地提醒你：各种编程的细节和技巧对你来说非常重要。通过本章的内容，你已经练习了许多计算技巧：各种变量的使用（包括 `vector` 和 `string`），算术和比较运算符的使用，选择和循环语句等。利用这些基本单元，你可以完成许多计算任务。你也练习了文本和数字的输入和输出，所有的输入和输出都可以表示为文本形式（包括图形）。利用一系列函数，你可以很好地组织你的程序。接下来你需要完成的任务是写好代码，即你的程序要正确、可维护和高效。更重要的是，你要踏踏实实地努力学习如何写好程序。

第5章 错 误

“我已经意识到从现在开始我的大部分时间将花在查找和纠正自己的错误上。”

——Maurice Wilkes, 1949

在本章中，我们将讨论程序的正确性、错误和错误处理。如果你是一个新手，你会发现这种讨论有时有点抽象，有时又显得过于细节化了。错误处理真的很重要吗？是的，它非常重要。在你写出别人愿意使用的程序前，你需要掌握几种错误处理方法。我们要做的是向你展示如何“像一个程序员一样思考”。它是建立在对细节和替代方案细致分析基础上的各种抽象策略的组合。

5.1 介绍

在前面章节的示例和练习中，我们已经多次提到了错误处理的相关内容，对于错误你应该已经有了一些初步的认识。在编写程序的时候，错误是不可避免的。当然，最后的程序必须是没有错误的，至少不存在我们不可接受的错误。

错误的分类有很多种，例如：

- 编译时错误：由编译器发现的错误。根据所违背的语法规则，编译时错误还可以进一步细分，例如：
 - 语法错误
 - 类型错误
- 连接时错误：当连接器试图将对象文件连接为可执行文件时发现的错误。
- 运行时错误：程序运行时发现的错误。运行时错误可以被进一步细分为以下几种：
 - 由计算机检测出的错误（硬件或操作系统）
 - 由库检测出的错误（例如标准库）
 - 由用户代码检测出的错误
- 逻辑错误：由程序员发现的会导致不正确结果的错误。

理想情况下，程序员的任务是消除所有的错误。但在实际中，这经常是不可行的。事实上，对于一个实际程序来说，如何准确定义“所有错误”都是很困难的。如果我们把一台正在执行程序计算机的电源线拔掉，那么这会是一种你认为的错误吗？在多数情况下，答案显然是否定的。但是，如果我们讨论的是医疗设备的监控程序或者电话交换机的控制程序的话，用户会认为包括程序在内的整个系统出了问题。用户只关心结果，而不关心导致这种情况的原因是计算机掉电，还是宇宙射线损坏了存放程序的存储器。因此，问题转化为“我们的程序能够检测到错误吗”。除非特别说明，我们会假定你的程序：

- 1) 对于所有合法输入应输出正确结果。
- 2) 对于所有非法输入应输出错误信息。
- 3) 不需要关心硬件故障。
- 4) 不需要关心系统软件故障。

5) 发现一个错误后, 允许程序终止。

不需要事先考虑假设 3)、4) 或 5) 情况的程序超出了本书的内容范围。但是, 假设 1) 和 2) 是属于程序员的基本专业能力范畴的, 而培养这种专业能力正是我们的目标之一。即使在实际中我们也不能 100% 达到理想目标, 但它是我们努力的方向。

在我们编写程序的时候, 出现错误是很自然的和不可避免的。问题是我们应该如何处理错误? 我们估计在开发正式软件时, 90% 以上的工作是放在如何避免、查找和纠正错误上。对于一些可靠性至关重要的程序来说, 这一比例甚至要更高。对于小程序来说, 你可以把错误处理做得很好。但是, 如果你很马虎, 你也可能做得很糟糕。

总之, 我们有下面三种方法来编写可接受的软件:

- 精心组织软件结构以减少错误。
- 通过调试和测试, 消除大部分程序错误。
- 确定余下的错误是不重要的。

上述任何一种方法都不能保证完全消除错误, 我们必须同时使用上述三种方法。

在开发可靠程序的时候, 经验往往会起巨大作用。这意味着, 根据用途的不同, 程序可以运行在可以接受的错误率下。请不要忘记, 理想情况下, 程序总是做正确的事。虽然我们一般只能接近这一理想情况, 但这不是我们不努力工作的借口。

5.2 错误的来源

错误的来源包括:

- 缺少规划: 如果没有事先规划好程序要做什么, 我们不可能充分检查所有“死角”, 并确认所有的可能情况都会被正确处理(即对任意输入程序都能给出正确结果或者充分的错误信息)。
- 不完备的程序: 在软件开发过程中, 显然会有一些我们没有考虑到的情况, 这是不可避免的。我们必须要达到的目标是掌握我们何时能够处理所有情况。
- 意外的参数: 函数会使用参数。如果为函数输入了一个不能处理的参数的话, 我们会遇到问题。例如, 为求平方根的标准库函数输入 -1.2 : $\text{sqrt}(-1.2)$ 。由于 $\text{sqrt}()$ 的输入和输出都是 `double`, 因此在这种情况下, 它不可能输出正确结果。5.5.3 节将讨论这种情况。
- 意外的输入: 典型的程序输入包括来自键盘、文件、图形用户界面和网络等的输入。对于这些输入, 程序一般都设定了许多前提假设。例如, 用户会输入一个数字。但是, 如果用户输入的是“喂, 闭嘴!”而不是期望的数字, 程序会怎样呢? 5.6.3 节和 10.6 节将讨论这类问题。
- 意外的状态: 多数程序都保留有很多系统各个部分所使用的数据(“状态”)。例如, 地址表、电话簿、温度记录等。如果这些数据是不完整的或者错误的, 那应该如何处理呢? 无疑程序的各个部分仍然应该正常运转。26.3.5 节将讨论这类问题。
- 逻辑错误: 这表示程序没有按照我们所期望的那样运行。我们不得不查找并修正这些问题。在 6.6 节和 6.9 节中我们将给出这类问题的例子。

上述列表可以用于实际开发。当我们开发一个软件的时候, 可以把上述列表作为检查表。在我们认为已经排除了所有潜在的错误来源之前, 软件是不能被交付使用的。实际上, 从项目开始时, 我们就始终要关注错误处理。因为, 没有认真考虑过错误处理的软件几乎是不可能正常工作

的,它还会被重新编写一遍。

5.3 编译时错误

在编写程序的时候,编译器是检查错误的第一道防线。在生成可执行文件之前,编译器通过分析代码来检查语法和类型错误。只有编译器认为代码完全符合语法规则,编译过程才能继续下去。编译器发现的大部分错误都很简单,属于“低级错误”。它们一般是由源代码的编辑错误导致的。其他一些问题则是由程序各个部分之间的交互引起的。作为初学者,你可能会觉得编译器很繁琐,但是当你学会使用一些语言特性(特别是类型系统)来直接表达你的思想时,你将会认识到编译器的错误检查功能的价值。如果没有编译器,乏味的除错工作将会花费你大量的时间。

举例来说,下面是一个简单函数的一些调用:

```
int area(int length, int width); // calculate area of a rectangle
```

5.3.1 语法错误

如果我们按照如下方式调用 `area()`,会有什么结果呢:

```
int s1 = area(7); // error: ) missing
int s2 = area(7) // error: ; missing
int s3 = area(7); // error: int is not a type
int s4 = area('7); // error: non-terminated character (' missing)
```

上面每一行程序都有一个语法错误,即它们不符合 C++ 语言的语法规则,因此编译器会拒绝它们。不幸的是,对你(即程序员)来说,理解语法错误的报告信息往往不是那么容易。为了确定错误,编译器往往会读取更多的信息。这会导致即使是一个小错误(往往在发现这个错误时,你会觉得不可思议,自己怎么会犯这种低级错误),编译器也会报告很多繁杂信息,甚至会指向程序中的其他行。因此,如果你在编译器所指向的错误行中没有发现错误的话,还应该检查一下前几行程序是否有错。

需要注意的是,编译器并不知道你想做什么。它只会报告你所做的是否有错,而不会报告你想做的是否有错。例如,在上面的例子中, `s3` 的声明有错,但是编译器不会告诉你:

“你拼错了 `int`, `i` 不要大写。”

而是报告如下信息:

“语法错误:标识符 '`s3`' 前丢失 '`;`'。”

“ '`s3`' 没有存储类型或标识符类型”。

“ '`int`' 没有存储类型或标识符类型”。

在你习惯并理解这些信息含义前,这些信息是很令人费解的。对于同一代码,不同的编译器可能会给出不同的错误信息。幸运的是,你会很快习惯理解这些信息的。实际上,上面这些令人费解的信息可以解释为:

“在 `s3` 前有一个语法错误,需要检查一下 `int` 或者 `s3` 的类型。”

实际上,发现这些问题并不是一件很困难的事。

试一试 尝试编译一下上面的例子,看看编译器的返回信息是什么。

5.3.2 类型错误

一旦排除了语法错误,编译器就会开始检查类型错误:它将会检查你所声明的变量和函数的类型(或者发现你忘记了声明类型),检查赋予变量或函数的数值或表达式的类型以及传递给函数

参数的数值或表达式的类型,例如:

```
int x0 = arena(7);           // error: undeclared function
int x1 = area(7);            // error: wrong number of arguments
int x2 = area("seven",2);    // error: 1st argument has a wrong type
```

让我们仔细分析一下这些错误:

1) 对于 `arena(7)`, 我们将 `area` 错写为 `arena`。因此编译器认为我们是要调用函数 `arena`。(编译器还有其他“想法”吗? 这就是前文所说的, 编译器是不知道你想做什么的。)假设没有函数 `arena()`, 你会得到未定义函数的错误信息。如果存在函数 `arena()` 并且这个函数能够接受 7 作为输入参数的话, 你会遇到一个更大的麻烦: 程序将会被正确编译, 但是它不会按照你预想的那样去运行(这是一个逻辑错误, 详见 5.7 节)。

2) 对于 `area(7)`, 编译器检查到的错误是参数个数不匹配。对于 C++ 语言, 函数调用必须使用正确的参数个数、参数类型和顺序。在合理使用类型检查系统时候, 它可以作为运行时的错误检查工具(详见 14.1 节)。

3) 对于 `area("seven", 2)`, 你可能期望编译器能够识别出 "seven" 表示的是数字 7, 但它做不到这点。当一个函数需要输入一个整数的时候, 我们不能给它一个字符串。C++ 确实支持一些隐含的类型转换(参见 3.9 节), 但不包括 `string` 到 `int` 的转换。编译器不会试图去猜测你所要表示的含义。不然, 你认为 `area("Hovellane", 2)`、`area("7", 2)` 和 `area("sieben", "zwei")` 表示什么含义呢?

上述只是一些简单的示例。编译器能够发现更多的错误信息。

试一试 尝试编译一下上面的例子, 看看编译器的返回信息是什么。尝试考虑一下你所遇到的更多的错误信息。

5.3.3 警告

当你使用编译器的时候, 你会希望它足够聪明, 能够理解你要表达的意思, 也就是说, 你可能会希望编译器报告的一些错误并不是真正的错误。这种想法是很自然的, 但令人惊奇的是, 当你有了一定编程经验后, 你会希望编译器能够拒绝更多代码, 而不是更少。看下面的例子:

```
int x4 = area(10,-7);        // OK: but what is a rectangle with a width of minus 7?
int x5 = area(10.7,9.3);     // OK: but calls area(10,9)
char x6 = area(100,9999);    // OK: but truncates the result
```

对于 `x4`, 我们没有从编译器得到错误信息。对于编译器来说, `area(10, -7)` 是正确的: `area()` 需要两个整型参数, 你给定了两个, 而没有人规定参数必须是正数。

对于 `x5` 来说, 好的编译器将给出警告信息: `double` 型参数 10.7 和 9.3 被截取为 `int` 型参数 10 和 9(参见 3.9.2 节)。然而, (旧的)语法规则认为可以隐式地将 `double` 转换为 `int`, 因此编译器不会拒绝函数调用 `area(10.7, 9.3)`。

对 `x6` 的初始化存在与 `area(10.7, 9.3)` 同样的问题。`area(100, 9999)` 的 `int` 型返回值, 即 999900, 被赋给一个 `char` 型变量。`x6` 最有可能的结果是“截取值”-36。同样, 一个好的编译器将会报告相关的警告信息, 即便(旧的)语法规则不拒绝相关代码。

当获得一定编程经验后, 你将学会如何脱离编译器去检查错误和避免编译器的弱点。但是, 不要过分自信: “程序已编译”并不意味着它能够运行。即使它能够运行, 在你消除逻辑错误前, 通常程序给出的也是错误的结果。

5.4 连接时错误

一个程序一般包括几个独立的编译部分, 称为翻译单元。程序中每一个函数在所有翻译单元

中的声明类型必须严格一致。我们使用头文件来保证这一点,详见8.3节。并且,程序中的每个函数只能定义一次。如果上述两条规则的任意一条被违反的话,连接器将报错。如何避免连接错误将在8.3节中讨论。下面是一个典型的程序连接错误示例:

```
int area(int length, int width);    // calculate area of a rectangle
```

```
int main()
{
    int x = area(2,3);
}
```

除非我们在另一个源文件中定义了 `area()`, 并且将该源文件编译得到的代码与当前代码连接, 否则连接器将报告没有找到 `area()` 的定义。

同时, `area()` 的定义必须与我们的调用具有严格相同的类型(包括返回值类型和参数类型), 即

```
int area(int x, int y) { /* ... */ }    // "our" area()
```

具有相同名称但是类型不同的函数将不会被匹配上, 并被忽略:

```
double area(double x, double y) { /* ... */ }    // not "our" area()
```

```
int area(int x, int y, char unit) { /* ... */ }    // not "our" area()
```

需要注意的是, 函数名的拼写错误并不总会导致连接错误。但是, 当遇到一个未定义函数被调用时, 编译器将立刻报告错误信息。这一点很好: 编译时错误早于连接时错误被发现有助于错误的及早排除。

正如上文所述, 函数的连接规则同样适用于程序的其他实体, 例如变量和类型: 具有同一名字的实体只能有一个定义, 但是可以有多个声明, 并且所有声明具有相同的类型。

5.5 运行时错误

如果你的程序没有编译时错误和连接时错误的话, 那么它就能运行了。现在乐趣才真正开始, 在你编写程序的时候, 你能够轻易地发现错误。但是, 对于运行程序时发现的错误, 你可能很难确定如何解决它, 例如:

```
int area(int length, int width)    // calculate area of a rectangle
{
    return length*width;
}

int framed_area(int x, int y)    // calculate area within frame
{
    return area(x-2,y-2);
}

int main()
{
    int x = -1;
    int y = 2;
    int z = 4;
    // ...
    int area1 = area(x,y);
    int area2 = framed_area(1,z);
    int area3 = framed_area(y,z);
    double ratio = double(area1)/area3;    // convert to double to get
                                           // floating-point division
}
```

在程序中,我们使用变量 x 、 y 、 z (而不是直接使用数值作为参数),这使得阅读代码的人和编译器更难发现问题。但是,这些调用会把负数赋给 `area1` 和 `area2`,导致面积表示为负数。我们能够接受这种明显违背数学和物理规律的错误结果吗?如果不能,应该由谁来检测这种错误:`area()`的调用者或者函数本身?这种错误又应该如何报告呢?

在回答这些问题之前,我们先看看上面代码中 `ratio` 的计算。这条语句看上去没有什么问题。但你是否发现了什么不对的地方了吗?如果没有,再仔细看看:`area3` 的值是 0,因此在 `double(area1)/area3` 中将除以 0。这会导致一个硬件检测错误并终止程序,同时报出一个与硬件相关的错误信息,这类错误就是运行时错误。如果你或者你的用户没有及时发现并解决这类错误的话,在程序运行时这类错误就可能出现。对于这类“硬件错误”,大多数人的容忍度都很低,因为他们不了解程序的细节而仅仅知道“某个地方出了问题”,这点信息对于处理问题帮助很小。在这种情况下,人们会很生气并对程序的提供者抱怨连连。

让我们再检查一下 `area()` 的参数错误问题。我们发现下面两种解决办法:

- 1) 让 `area()` 的调用者来处理不正确的参数。
- 2) 让 `area()` (被调用函数)来处理不正确的参数。

5.5.1 调用者处理错误

先看看第一种方法(“让用户意识到问题”)。如果 `area()` 是一个由我们不能修改的库提供的函数,我们将选择这种方法。不管怎样,这都是一个合适的选择。

在 `main()` 函数中保护 `area(x, y)` 的调用是很容易的:

```
if (x<=0) error("non-positive x");
if (y<=0) error("non-positive y");
int area1 = area(x,y);
```

的确,当你发现一个错误后,唯一的问题就是如何解决它。这里我们调用了 `error()` 函数,假设它能够做一些错误处理工作。实际上,在 `std_lib_facilities.h` 中我们提供了一个 `error()` 函数,它能够终止程序运行并将字符串参数作为系统错误信息输出。如果你希望输出自己的错误信息并做其他的操作,参看 `runtime_error` (参见 5.6.2 节、7.3 节、7.8 节和附录 B.2.1)。对于初学者来说,这已经足够了,它还可以作为更复杂错误处理的实例。

如果我们不需要明确区分每一个参数,我们还可以简化程序如下:

```
if (x<=0 || y<=0) error("non-positive area() argument"); // || means "or"
int area1 = area(x,y);
```

为了对 `area()` 的参数实现完全保护,我们需要使用 `framed_area()` 函数,程序改写为:

```
if (z<=2)
    error("non-positive 2nd area() argument called by framed_area()");
int area2 = framed_area(1,z);
if (y<=2 || z<=2)
    error("non-positive area() argument called by framed_area()");
int area3 = framed_area(y,z);
```

这看上去有些混乱,而且还存在一些基本问题。上面的程序只有在我们确切了解 `framed_area()` 如何使用 `area()` 的情况下才能编写正确。要知道 `framed_area()` 对每一个参数都减了 2。我们不得不了解这么多细节情况。如果有人把 `framed_area()` 修改为减 1 而不是 2,我们又该怎么办呢?如果这种情况发生,我们不得不查找程序中的每一个 `framed_area()` 调用,并做相应的修改。这称为“易碎”代码,因为它很容易被破坏。这也是一个“魔数”的例子(参见 4.3.1 节)。为了减少程序的“易碎性”,我们可以在 `framed_area()` 中用一个命名常量代替具体的数值:

```

const int frame_width = 2;
int framed_area(int x, int y) // calculate area within frame
{
    return area(x-frame_width,y-frame_width);
}

```

这个常量也可以被 framed_area() 的调用者使用:

```

if (1-frame_width<=0 || z-frame_width<=0)
    error("non-positive argument for area() called by framed_area()");
int area2 = framed_area(1,z);
if (y-frame_width<=0 || z-frame_width<=0)
    error("non-positive area() argument called by framed_area()");
int area3 = framed_area(y,z);

```

仔细看看上面的代码,你能确定它是正确的吗?它形式上漂亮吗?它易读吗?事实上,我们发现它很糟糕(因此容易出错)。我们将代码长度增加了三倍,并且还不得不去了解 framed_area() 的实现细节。但是我们仍然没有达到目的,应该有更好的办法解决这一问题!

再看看原始代码:

```

int area2 = framed_area(1,z);
int area3 = framed_area(y,z);

```

这段代码也许会出错,但我们至少知道它要做什么。如果把错误检查移到 framed_area() 内部,我们可以保留这段代码。

5.5.2 被调用者处理错误

在 framed_area() 内部实现错误检查非常简单, error() 仍然可以被用于错误报告:

```

int framed_area(int x, int y) // calculate area within frame
{
    const int frame_width = 2;
    if (x-frame_width<=0 || y-frame_width<=0)
        error("non-positive area() argument called by framed_area()");
    return area(x-frame_width,y-frame_width);
}

```

这一实现非常好,而且我们也不用为每一个 framed_area() 调用写测试。在一个大程序中,对一个会被调用 500 次的常用函数来说,这一点非常有用。而且,如果需要对错误处理进行修改的话,我们只需要在一个地方进行改动就可以了。

需要注意的是,在这里我们很自然地“从调用者必须检查参数”的方法转变到“函数必须检查自己的参数”的方法(也称为“被调用者检查”)。后者的好处在于参数检查只在一个地方实现。我们不需要在整个程序中查找调用点。而且,参数检查只在一个地方实现,我们可以方便地掌握参数检查的全部信息。

让我们把这一思想应用到 area() 中:

```

int area(int length, int width) // calculate area of a rectangle
{
    if (length<=0 || width <=0) error("non-positive area() argument");
    return length*width;
}

```

上面程序实现了对于 area() 调用的所有错误处理,因此我们不再需要调用 framed_area()。进一步改进,我们可能需要对于错误信息的更准确描述。

函数的参数检查看上去很简单,但是为什么很多人还会忽视它呢?不注意错误处理是一个原因,粗心大意是另一个原因。此外,还有许多其他因素:

- 我们不能改变函数定义:在函数库内定义的函数因某种原因不能被改变。原因可能是该

函数也被其他程序调用,相关的错误处理可能不一致。也可能是该函数为其他人所有,你没有相关源代码。也可能是该函数属于某一个会定期更新的库,如果你修改了原函数,那么当库更新时,你不得不再次修订它。

- 被调函数不知道应该如何处理错误:这是库函数的典型应用。库的作者可以检测到错误,但是只有你才知道应该如何处理错误。
- 被调函数不知道它是在哪里被调用的:当你获得一个错误信息后,它会告诉你某个错误发生了,但不会告诉你程序是如何执行到这个点上的。但有时候,你会需要更详细地了解错误信息。
- 性能:对于小函数来说,错误检查的代价可能会超过计算本身。例如,对于 `area()` 函数来说,错误检查代价超过函数本身两倍(这里指的是需要执行的机器指令的数量,而不是源代码长度)。对某些程序来说,这很重要,特别是当函数之间相互调用的时候,相关的参数变化不大,但是参数检查会被反复执行。

那我们应该怎么做呢?除非你有足够的理由,否则参数检查还是应该在函数内部完成。

在介绍一些相关内容后,我们将在 5.10 节中继续讨论如何处理错误参数的问题。

5.5.3 报告错误

让我们考虑另外一个问题:在检查一系列参数后,一旦发现了错误,你应该如何做?有时,你可以返回一个“错误值”,例如:

```
// ask user for a yes-or-no answer;
// return 'b' to indicate a bad answer (i.e., not yes or no)
char ask_user(string question)
{
    cout << question << "? (yes or no)\n";
    string answer = " ";
    cin >> answer;
    if (answer == "y" || answer == "yes") return 'y';
    if (answer == "n" || answer == "no") return 'n';
    return 'b'; // 'b' for "bad answer"
}

// calculate area of a rectangle;
// return -1 to indicate a bad argument
int area(int length, int width)
{
    if (length <= 0 || width <= 0) return -1;
    return length*width;
}
```

如上面示例所示,我们可以让被调函数进行详细检查,同时让调用者按需要处理错误。看上去这种方法是可行的,但在某些情况下,这种方法会带来很多问题使得它实际上不能被接受:

- 所有调用者和被调函数都需要进行错误检查。调用者要进行的检查可能很简单,但还必须要编写这段代码,并决定在错误发生时如何处理。
- 调用者可能会忘记做错误检查。这可能导致程序在运行时出现不可预测的问题。
- 许多函数并没有可以用作标记错误信息的额外返回值。例如,一个用于从输入设备读入整数的函数(例如 `cin` 的操作符 `>>`),其返回值可以是任意整数,因此不能用一个专门的整数来表示错误信息。

上面程序中的第二种情况显示的就是调用者忘记了进行测试,这会导致某些不可预见的问题,例如:

```
int f(int x, int y, int z)
{
    int area1 = area(x,y);
    if (area1<=0) error("non-positive area");
    int area2 = framed_area(1,z);
    int area3 = framed_area(y,z);
    double ratio = double(area1)/area3;
    //...
}
```

你看出错误在哪里了吗？问题就是缺少了错误检查。因为没有明显的错误代码，这类错误往往很难被发现。

试一试 测试程序的不同输入和返回值。输出函数 `area1`、`area2`、`area3` 和 `ratio` 的值。

尝试插入各种测试程序直到所有错误都被检测到。如何才能知道所有错误都被找到了呢？这不是一个脑筋急转弯问题，在本例中，你可以通过输入有效的参数检测所有的错误。

还有另外一种解决这一问题的方法：使用异常处理。

5.6 异常

与大多数现代编程语言类似，C++ 也提供了一种错误处理机制：异常。为了保证检测到的错误不会被遗漏，异常处理的基本思想是把错误检测（在被调函数中完成）和错误处理（在主调函数中完成）分离。异常提供了一条可以把各种最好的错误处理方法组合在一起的途径。错误处理很繁琐，但异常可以让它变得简单一些。

异常的基本思想是：如果函数发现一个自己不能处理的错误，它不是正常返回，而是抛出异常来表示错误的发生。任何一个直接或间接的函数调用者都可以捕捉到这一异常，并确定应该如何处理。函数可以用 `try` 语句（细节情况将在后面小节中介绍）来处理异常：把所要处理的异常情况罗列在 `catch` 语句后。如果出现一个没有被任何调用函数处理的异常，程序终止运行。

在后续章节中（参见第 19 章），我们还将介绍异常的一些高级用法。

5.6.1 错误参数

下面是函数 `area()` 使用异常处理的版本：

```
class Bad_area { };    // a type specifically for reporting errors from area()

// calculate area of a rectangle;
// throw a Bad_area exception in case of a bad argument
int area(int length, int width)
{
    if (length<=0 || width<=0) throw Bad_area();
    return length*width;
}
```

如果参数正确，我们会返回计算的面积；否则结束函数 `area()`，并抛出异常，希望这个异常能够被捕获并做出相应错误处理。`Bad_area` 是我们定义的一个新类型。它的目的是作为函数 `area()` 中异常的标识，以便被捕获时能够确认异常来自哪里。用户自定义类型（类和枚举）将在第 9 章讨论。需要注意的是，`Bad_area()` 表示“定义一个名为 `Bad_area` 类型的对象”。因此，`throw Bad_area()` 表示“定义一个名为 `Bad_area` 类型的对象并抛出它”。

现在我们可以这样写：


```

int main()
try {
    int x = -1;
    int y = 2;
    int z = 4;
    // ...
    int area1 = area(x,y);
    int area2 = framed_area(1,z);
    int area3 = framed_area(y,z);
    double ratio = area1/area3;
}
catch (Bad_area) {
    cout << "Oops! bad arguments to area()\n";
}

```

首先要注意的是，上面的错误处理针对的是所有对 `area()` 的调用，包括主函数里的一个调用和两个通过 `framed_area()` 的间接调用。其次，注意如何处理错误与检测错误是分离的：`main()` 不知道哪个函数做了 `throw Bad_area()` 动作，`area()` 不知道哪个函数会捕捉它所抛出的 `Bad_area` 异常。对于使用了许多库的大程序来说，这一分离非常重要。因为在编程时没有人希望同时对应用程序和库代码进行修改，所以没有人能够“通过在正确位置简单增加几行代码来修正错误”。

5.6.2 范围错误

大多数实际程序都需要处理数据集合，即会用各种类型的数据表、队列等来完成某项任务。在 C++ 语言中，我们一般把“数据集合”称为容器。最常用的标准库容器是 4.6 节介绍的 `vector`。一个 `vector` 中包含一组数据，我们可以通过 `vector` 的成员函数 `size()` 来获得数据的个数。如果我们引用了一个不在有效范围 `[0:v.size())` 内的下标，会出现什么情况呢？需要注意的是，`[low:high)` 表示从 `low` 到 `high - 1` 的下标范围，它包括 `low` 但不包括 `high`，如右图所示。



在回答上面的问题前，我们先看看另外一个问题和它的答案：

“为什么要这么做呢？”毕竟，你应该明白下标只能在范围 `[0:v.size())` 内，因此确保这一点就可以了呀！

话虽然是这么说的，但是实际上，很难保证这种情况不会发生。看看下面这个似乎合理的程序：

```

vector<int> v;    // a vector of ints
int i;
while (cin>>i) v.push_back(i);    // get values
for (int i = 0; i<=v.size(); ++i)    // print values
    cout << "v[" << i << "] == " << v[i] << endl;

```

你看出问题了吗？试着把它标识出来。这不是一个一般性的错误。这种错误往往是由我们自身的原因导致的，特别是在工作到很晚我们很累的时候。当我们很劳累或者很繁忙的时候，这类错误经常会出现。在我们对 `v[i]` 进行操作时候，我们用 0 和 `size()` 来保证 `i` 总是在合法的范围内。

不幸的是，我们犯了一个错误。仔细看看 `for` 循环：它的终止条件是 `i <= v.size()` 而不是 `i < v.size()`。这会导致一个不幸的结果：如果我们读入了 5 个整数，它会输出 6 个结果。因为我们试图读 `v[5]` 的值，而它已经超出了 `vector` 的存储空间范围。这类错误是非常普遍的而且很“出名”，人们为它起了很多名字：偏一位错误 (off-by-one error)；范围错误 (range error)，因为下标不在 `vector` 的合法值范围内；边界错误 (bounds error)，因为下标不在 `vector` 的合法值范围内。

下面是一个具有同样问题的简单版本：

```

vector<int> v(5);
int x = v[5];

```

然而，我们还是怀疑你没有认识到这个问题的真实性和严重性。

当我们犯了这个错误后，实际会发生什么情况呢？vector 的下标操作知道元素的个数，因此它可以检测是否出错（我们所使用的 vector 也是可以的，参见 4.6 节和 19.4 节）。如果检查到错误，下标操作将抛出一个名为 `out_of_range` 的异常。因此，如果程序的代码有范围错误并导致了异常的话，我们至少能够捕捉到一个异常信息。

```
int main()
try {
    vector<int> v;           // a vector of ints
    int x;
    while (cin>>x) v.push_back(x);    // set values
    for (int i = 0; i<=v.size(); ++i) // print values
        cout << "v[" << i << "] == " << v[i] << endl;
} catch (out_of_range) {
    cerr << "Oops! Range error\n";
    return 1;
} catch (...) {             // catch all other exceptions
    cerr << "Exception: something went wrong\n";
    return 2;
}
```

需要注意的是，范围错误可以被认为是 5.5.2 节中提到的参数错误的一个特例。我们不能保证自己对向量下标的范围检查总是正确的，因此我们让向量的下标操作来做这一检查。正如上文所述，向量的下标函数（名为 `vector::operator[]`）能够通过抛出异常来报告错误。它还能做什么呢？如果发生一个范围错误，它是不知道我们将会如何处理的。向量的编写者甚至不知道向量是属于程序的哪一段代码。

5.6.3 输入错误

我们将把处理输入错误的细节讨论放到 10.6 节。不过，一旦输入错误被发现，利用与处理参数错误和范围错误相同的技术，它将会被迅速处理。这里，我们只展示如何判断输入是否正确，下面是输入一个浮点数的情况：

```
double d = 0;
cin >> d;
```

通过测试 `cin`，我们可以确定最后一个输入操作是否成功：

```
if (cin) {
    // all is well, and we can try reading again
}
else {
    // the last read didn't succeed, so we take some other action
}
```

有几种原因可能会导致输入操作失败。其中一个原因就是 `>>` 操作输入的不是所要求的 `double` 类型数据。

在开发工作的早期，我们主要关注于发现错误但并没有给出特别好的办法来解决它。我们做的仅仅是报告错误并终止程序。下面，我们将尝试更好的办法来处理它。例如：

```
double some_function()
{
    double d = 0;
    cin >> d;
    if (!cin) error("couldn't read a double in 'some_function()'");
    // do something useful
}
```

传递给函数 `error()` 的字符串将被输出，它可以作为调试的有益帮助或反馈给用户的信息。这个对

于很多程序都很有用的 `error()` 应该如何编写呢？因为我们不知道应该如何处理返回值，所以这个函数没有返回值，它在输出信息后将直接终止程序。此外，在终止程序前，我们可以做一些次要的操作，例如保持窗口一段足够长的时间以便我们阅读信息。显然，这是异常处理应该做的工作（参见 7.3 节）

标准库定义了一些异常，例如 `vector` 的 `out_of_range`。此外，标准库还提供 `runtime_error` 异常。`runtime_error` 对我们非常有用，因为它包含一个字符串，可以用于错误处理。有了它，`error()` 可以写成下面的形式：

```
void error(string s)
{
    throw runtime_error(s);
}
```

当我们想处理 `runtime_error` 的时候，我们捕捉到它就可以了。对于简单程序来说，在 `main()` 中捕捉 `runtime_error` 更理想：

```
int main()
try {
    // our program
    return 0;    // 0 indicates success
}
catch (runtime_error& e) {
    cerr << "runtime error: " << e.what() << '\n';
    keep_window_open();
    return 1;    // 1 indicates failure
}
```

对 `e.what()` 的调用将从 `runtime_error` 中提取错误信息。在下面语句

```
catch(runtime_error& e) {
```

中的 `&` 表示我们希望“以引用方式传递异常”。现在，我们暂时把它看做一种不相关的技术。在 8.5.4 节 ~ 8.5.6 节中，我们会详细解释通过引用传递参数的含义。

注意，这里我们使用 `cerr` 而不是 `cout` 进行错误信息输出：`cerr` 与 `cout` 用法相同，只是它是专门用于错误信息输出的。默认情况下，`cerr` 和 `cout` 都输出到屏幕上，但是 `cerr` 没有经过优化，所以更适合错误信息输出，在一些操作系统中它还可以被转到其他输出目标，例如一个文件中。使用 `cerr` 也有助于我们编写与错误相关的文档，因此，我们使用 `cerr` 作为错误信息输出。

显然，`out_of_range` 与 `runtime_error` 不同。当捕获到 `runtime_error` 异常时候，错误处理方式不会像处理 `out_of_range`（向量或者其他标准库容器导致的）那样。但是，`out_of_range` 与 `runtime_error` 都是“异常”，我们可以对异常进行一些通用的处理：

```
int main()
try {
    // our program
    return 0;    // 0 indicates success
}
catch (exception& e) {
    cerr << "error: " << e.what() << '\n';
    keep_window_open();
    return 1;    // 1 indicates failure
}
catch (...) {
    cerr << "Oops: unknown exception!\n";
    keep_window_open();
    return 2;    // 2 indicates failure
}
```

这里我们加上 `catch(...)` 来处理任何其他类型的异常。

我们用来同时处理 `out_of_range` 与 `runtime_error` 两种异常的是一个单一类型 `exception`，它称为两者的公共基类(超类型, `supertype`)。这是一种非常有用的通用技术，我们将在第 13 章 ~ 第 16 章中详细介绍。

需要注意的是，`main()` 的返回值传递给了调用程序的“系统”。一些系统(例如 UNIX)经常会用到这些返回值，而另一些系统(例如 Windows)会忽略它们。返回值为 0 表示 `main()` 成功完成，而非 0 返回值表示某些错误发生了。

在使用 `error()` 的时候，你可能希望能同时传递两部分信息来描述所发生的问题。在这种情况下，我们可以把这两部分信息连接起来作为一个字符串传递。因此，我们提供了第二个版本的 `error()`：

```
void error(string s1, string s2)
{
    throw runtime_error(s1+s2);
}
```

在我们的需求极大提高，以及我们作为设计师和程序员的水平相应提高之前，这种简单的错误处理方式就够用了。需要注意的是，`error()` 的使用与程序执行路径上有多少函数调用是无关的：`error()` 会查找距离最近的捕获 `runtime_error` 的操作，这个操作通常就在 `main()` 中。使用异常和 `error()` 的例子，可以参考 7.3 节和 7.7 节的内容。如果某个异常没有被捕获到，默认情况下，你会得到一个系统错误(“未捕获异常”错误)。

试一试 尝试看看未捕获异常错误会是什么样：运行一个使用了 `error()` 而没有捕获任何异常的小程序。

5.6.4 截断错误

在 3.9.2 节中，我们曾看过一个故意的错误：当我们给一个变量赋了一个“太大”的值后，这个值会被截断。例如：

```
int x = 2.9;
char c = 1066;
```

这里 `x` 的值是 2 而不是 2.9，因为 `x` 是整型，而整型没有小数部分，只有整数部分(这是显然的)。类似地，如果我们使用 ASCII 字符集，`c` 的值将是 42(表示字符 `*`)，而不是 1066，因为字符集中没有值为 1066 的字符。

在 3.9.2 节中，我们已经看到如何通过测试来确保截断错误不发生。有了异常(和模板，参见 19.3 节)，我们可以编写函数来测试能够引起值改变的赋值或初始化操作，有错误发生时，抛出 `runtime_error` 异常。例如：

```
int x1 = narrow_cast<int>(2.9);    // throws
int x2 = narrow_cast<int>(2.0);    // OK
char c1 = narrow_cast<char>(1066); // throws
char c2 = narrow_cast<char>(85);   // OK
```

这里 `<...>` 的用法与 `vector<int>` 中的尖括号相同。当需用确定一个类型而不是数值时，我们可以这样使用。它们被称为模板参数。当需要进行类型转换而不确定数值“是否适合”目标类型时，我们可以使用 `narrow_cast`。它是在 `std_lib_facilities.h` 中定义并用 `error()` 实现的。单词 `cast` 的意思是“类型转换”，它暗示进行转换的对象是有问题的(就像对一条断腿固定石膏模一样)。需要注意的是，类型转换并不会改变操作数，而是生成了一个与其操作数对应的新数值，其类型在 `<...>` 中指明。

5.7 逻辑错误

在解决了开始的编译器和连接器错误后,程序就能运行了。通常情况下,此时的程序要么没有输出要么输出结果是错误的。产生这样结果的原因有很多,具体原因可能是你所理解的程序逻辑是错误的;可能你所编写的程序并不是你所设想的;可能你写控制语句时犯了“低级错误”;或者其他原因。通常,逻辑错误是最难被发现和排除的,因为在这种情况下计算机所做的正是你让它做的事情。你此时的任务是要发现你让计算机做的事情为什么没有反映你的真实意愿。基本上,我们可以认为计算机是一个速度非常快的笨蛋。它只是精确地完成你让它做的事情,这一点有时会让人感到很尴尬。

让我们通过一个简单的例子来理解逻辑错误。下面的代码在一组数据中找出最低、最高和平均温度:

```
int main()
{
    vector<double> temps;    // temperatures

    double temp = 0;
    double sum = 0;
    double high_temp = 0;
    double low_temp = 0;

    while (cin>>temp)        // read and put into temps
        temps.push_back(temp);

    for (int i = 0; i<temps.size(); ++i)
    {
        if(temps[i] > high_temp) high_temp = temps[i];    // find high
        if(temps[i] < low_temp) low_temp = temps[i];      // find low
        sum += temps[i];    // compute sum
    }

    cout<< "High temperature: " << high_temp<< endl;
    cout<< "Low temperature: " << low_temp << endl;
    cout<< "Average temperature: " << sum/temps.size() << endl;
}
```

为了测试上述程序,我们输入 2004 年 2 月 16 日得克萨斯州拉伯克(Lubbock)气象中心测得的每小时温度值(得克萨斯州使用的是华氏温度)。

```
-16.5, -23.2, -24.0, -25.7, -26.1, -18.6, -9.7, -2.4,
 7.5, 12.6, 23.8, 25.3, 28.0, 34.8, 36.7, 41.5,
40.3, 42.6, 39.7, 35.4, 12.6, 6.5, -3.7, -14.3
```

输出是:

```
High temperature: 42.6
Low temperature: -26.1
Average temperature: 9.3
```

初学者会认为上面的程序是没有问题的,不负责任的程序员会把它直接交付用户。谨慎的做法应该用另外一组数据再次测试程序,这组数据来自 2004 年 7 月 23 日。

```
76.5, 73.5, 71.0, 73.6, 70.1, 73.5, 77.6, 85.3,
88.5, 91.7, 95.9, 99.2, 98.2, 100.6, 106.3, 112.4,
110.2, 103.6, 94.9, 91.7, 88.4, 85.2, 85.4, 87.7
```

这一次,输出结果是:

High temperature: 112.4
 Low temperature: 0.0
 Average temperature: 89.2

哦，一定是什么地方出问题了。7月份的拉伯克出现严寒(0.0华氏度大约是零下18摄氏度)将意味着世界末日！你能找出错误在哪里吗？原因在于low_temp的初值是0.0，除非有一个温度值低于0.0，否则它将一直是0.0。

试一试 运行上面的程序，检验一下输入数据确实会产生那样的结果。尝试一下利用其他输入数据“打破”程序(即令程序输出错误结果)。看看你需要输入多少组数据，才会令程序出错？

不幸的是，程序中还有其他错误。如果所有温度值都低于0的话，会发生什么？high_temp的初始化与low_temp存在同样的问题：除非有一个温度值高于0.0，否则high_temp将始终是0.0。在南极，这个程序同样会出现问题。

这类错误是相当典型的，在程序编译的时候它不会出错，对于“合理的”输入也不会出错。但是，我们忘记了仔细思考应该将哪些数据定义为“合理的”。这个程序的改进如下：

```
int main()
{
    double temp = 0;
    double sum = 0;
    double high_temp = -1000; // initialize to impossibly low
    double low_temp = 1000; // initialize to "impossibly high"
    int no_of_temps = 0;

    while (cin >> temp) { // read temp
        ++no_of_temps; // count temperatures
        sum += temp; // compute sum
        if (temp > high_temp) high_temp = temp; // find high
        if (temp < low_temp) low_temp = temp; // find low
    }

    cout << "High temperature: " << high_temp << endl;
    cout << "Low temperature: " << low_temp << endl;
    cout << "Average temperature: " << sum/no_of_temps << endl;
}
```

这个程序正确吗？你如何确定？你应该如何准确定义程序的“正确性”？哪里的温度值会达到1000和-1000呢？想一想关于“魔数”的警告(见5.5.1节)。程序中使用1000和-1000这样的文字常量不是一种好的编程风格，但这两个值也会有问题吗？是否有哪个地方的温度会低于华氏-1000度(-573摄氏度)呢？是否有哪个地方的温度会高于1000华氏度(538摄氏度)呢？

试一试 查阅一下相关资料，为我们程序的min_temp(“最低温度”)和max_temp(“最高温度”)设定合适的常量值。这些常量将决定我们程序的适用范围。

5.8 估计

想象一下，你已经编写了一个进行简单计算的程序，例如计算六边形面积。运行这个程序你得到的结果是-34.56。你知道它肯定有问题，为什么？因为面积不可能是负数。接下来，你找到并修改了相应错误(不管它是什么)，然后再次得到一个结果21.65685。这一次对了吗？很难说，因为我们一般不能马上说出计算六边形面积的公式。为了避免将一个产生可笑结果的程序交付用户而使我们出丑，我们应该在交付之前检查程序结果是否合理。在本例中，这个工作很简单。六边形与正方形很接近，在纸上画一个六边形，目测一下，它接近一个3乘3的正方形，这样的一

个正方形的面积是 9。真倒霉,结果 21.65685 不可能是对的。因此我们继续修改程序,新程序的计算结果为 10.3923。这一次,它可能是正确的了!

这种方法与六边形无关,其关键思想是,除非我们知道正确的结果大概是什么,或者与什么比较类似,否则我们不能判定得到的结果是否合理。要不断问自己如下问题:

1) 这个问题的答案合理吗?

还应该问一个更一般的(通常也更难的)问题:

2) 我们应该如何判定一个结果是否合理?

这里,我们没有问“最准确的答案是什么?”或“正确答案是什么?”这是我们编写的程序要告诉我们的。我们所要知道的就是这个答案是合理的。只有确定了结果的合理性后,我们才能做下一步工作。

估计(estimation)是一种优雅的艺术,它将常识与一些用来解决常见问题的非常简单的数学方法结合起来。一些人很擅长在头脑中估计,但我们更提倡在纸上随意写写画画,因为这种方法能够帮助我们减少错误。我们这里所说的估计是一种非正式的技术。有时候它会被(幽默地)称为瞎估计(guessimation),因为它是将一点猜测和一点计算结合起来的方法。

试一试 我们的六边形的每边长都是 2 厘米。得到的结果是正确的吗?亲自动手“在信封背面”算一下。在纸上把它画出来,不要觉得这太简单了。许多著名的科学家都有这样一种令人敬佩的能力:用笔在信封背面(或餐巾纸上)估计出问题的近似结果。这是一种能力,实际上也是一种简单的习惯,它能够帮助我们节省很多时间并减少很多错误。

通常,估计的过程包括对正确计算所需输入数据的估计,我们还未对此进行过讨论。假定我们要测试一个估计城市间驾驶时间的程序,开车从纽约到丹佛花费 15 小时或 33 分钟,这个时间是否合理呢?从伦敦到尼斯呢?为什么合理或者为什么不合理呢?在回答这些问题时,你需要用什么数据来估计行车时间呢?通常,在互联网上快速搜索一下是最有用的方法。例如,2000 英里是纽约和丹佛之间距离的合理估计。对于驾驶汽车来说,保持每小时 130 英里的平均速度是很困难的(或者是非法的)。因此 15 小时的结果是不合理的($15 * 130$ 略小于 2000)。你可以检验一下:我们既高估了距离,也高估了平均速度,但在检验合理性时,我们不需要非常准确,只要估计得差不多就可以了。

试一试 估计一下上述开车时间。同时,也估计一下相应的飞行时间(假定乘坐普通的民航航班)。然后,利用更准确的数据来验证你的估计,例如地图和时刻表。我们建议使用互联网资源。

5.9 调试

当你写完一个程序后,它会有不少错误。小程序偶尔会一次通过,但如果一个复杂的大程序也出现这种情况的话,你一定要保持一个非常谨慎的态度。如果它确实第一次运行就正确的话,赶紧告诉你的朋友们来庆祝一下吧,因为这种事情不是每年都有的。

因此,当你写完代码后,你要做的就是找到并排除错误。这一过程称为调试,错误被称为 bug。据说 bug(有昆虫的意思)一词来自于早期真空管计算机时代,那时的计算机是占据了很大空间的真空管计算机,当小昆虫进入电路板后就会导致硬件故障。一些人将 bug 一词借用过来专指软件中的错误,其中最著名的就是 Grace Murray Hopper, COBOL 编程语言的发明人(参见

22.2.2.2 节)。这个词第一次出现已经是五十多年以前的事情了，现在它已经被人们普遍接受了。查找并排除错误的过程称为调试。

调试可以简单地描述为：

- 1) 让程序编译通过。
- 2) 让程序正确连接。
- 3) 让程序完成我们希望它做的工作。

基本上，我们要一次次重复这个过程：对于大程序，需要上百次、上千次、年复一年。每当程序不能正常工作，我们就要找出问题并修正。我认为在编程中调试是最乏味、最费时间的工作，因此应该不遗余力地做好设计和编码工作，以使除错的时间降到最低。有的人会在调试过程中体验到搜寻 bug 和深入程序精髓的快乐——调试可以和视频游戏一样让人上瘾，会把程序员没日没夜地黏在计算机前（以个人的经验，我可以证实这一点）。

下面是尽量避免调试的方法：

```
while (the program doesn't appear to work) {    // pseudo code
    Randomly look through the program for something that "looks odd"
    Change it to look better
}
```

为什么我们要提到这些呢？显然，糟糕的方法会使成功的机会变得很低。不幸的是，上述行为恰好概括了许多人在深更半夜工作，特别是毫无头绪时所做的“无用功”。

调试中的关键问题是：

我如何知道程序是否真正运行正确呢？

如果你不能回答这个问题，你将会陷入长时间、乏味的调试工作中，而且你的用户也很可能陷入麻烦。我们会反复强调这个问题，任何有助于回答此问题的信息都会减少调试的工作量，并有助于生成正确、可维护的程序。实际上，我们宁愿程序设计良好，使得错误无处藏身。一般来说这个目标很难达到，但我们还是会在程序设计上下功夫，以最小化出错的概率，同时最大化发现错误的概率。

5.9.1 实用调试技术

在编写代码前一定要仔细考虑调试问题。如果已经写完很多行代码之后你才想到应该如何简化调试问题的话，那就太晚了。

首先你要决定如何报告错误：“使用 `error()` 并在 `main()` 中捕获异常”应该是你从本书学到的一个标准答案。

要提高程序的易读性，这样你会有更多机会发现错误所在：

- 为代码做好注释。这并不意味着“加上大量注释”。能靠代码本身表达清楚的，不要用注释。注释的内容应该是你不能在代码里说清楚的部分。你应该用尽量简洁、清楚的语言把它们说清楚以下内容：
 - 程序的名称
 - 程序的目的
 - 谁在什么时候写了这个代码
 - 版本号
 - 代码的每部分的目的是什么
 - 总体设计思想是什么

- 源代码是如何组织的
- 输入数据的前提假设是什么
- 还缺少哪一部分代码，程序还不能处理哪些情况
- 使用有意义的名字。
 - 这并不意味着使用“长名字”。
- 使用一致的代码层次结构。
 - 你是代码的负责人，集成调试环境(IDE)可以帮助但不能替代你做所有事情。
 - 本书所使用的编程风格可以作为一个有益的起点。
- 代码应该被分成许多小函数，每个函数表达一个逻辑功能。
 - 尽量避免超过一或两页的函数；大多数函数应该很短。
- 避免使用复杂的程序语句。
 - 尽量避免使用嵌套的循环、嵌套的 if 语句、复杂的条件等。不幸的是，有时你必须这样做，但请记住复杂代码是最容易隐藏错误的地方。
- 在可能的情况下，使用标准库而不是你自己的代码。
 - 同样是完成某个功能，标准库一般会比你自己的程序考虑得更周全，并且经过了更完备的测试。

上面的描述有些抽象，但在后续的篇幅中，我们会通过一个个的例子来向你详细解释。

程序首先要编译通过。在这个阶段，编译器显然是你最好的助手。它给出的错误信息通常是很有用的，虽然我们总希望得到更准确的信息，除非你是一个真正的专家，否则你还是假定编译器是正确的为好。如果你是一个真正的专家，这本书不是为你写的。有时，你会觉得编译器遵循的规则实在是太愚蠢而且没有必要（通常并非如此），这些规则能够而且应该更简单（的确，但是它们不是这样的）。然而，俗话说“糟糕的工匠才会抱怨他的工具”。好的工匠了解自己的工具的长处和短处，并能在此基础上对自己的工作进行相应调整。下面是一些常见的编译时错误：

- 每一个字符串常量都用双引号终止了吗？


```
cout << "Hello, " << name << "\n";    // oops!
```
- 每一个字符常量都用单引号终止了吗？


```
cout << "Hello, " << name << "\n";    // oops!
```
- 每一个程序块都终止了吗？


```
int f(int a)
{
    if (a>0) { /* do something */ else { /* do something else */ }
}    // oops!
```
- 所有圆括号都一对一匹配吗？


```
if (a<=0    // oops!
    x = f(y);
```

编译器一般会“稍晚些”报告这类错误，因为它不知道你本来想在 0 之后输入一个右括号。

- 每一个名字都声明了吗？
 - 你是否包含了所有必需的头文件（目前，应该用 `#include "std_lib_facilities.h"`）？
 - 所有名字都在使用之前进行声明了吗？
 - 你是否拼正确拼写了所有名字？


```
int count; /* ... */ ++Count;    // oops!
char ch;   /* ... */ Cin>>c;    // double oops!
```

- 你用分号终止了每个表达式语句吗？

```
x = sqrt(y)+2    // oops!
z = x+3;
```

在本章的简单练习中，我们将给出更多的例子。同时，请记住 5.2 节中提出错误的分类。

在程序的编译和连接完成后，下一步就是最难的部分了：找出程序没有按照我们的意图去运行的原因。你要检查输出结果，并尽力搞清楚为什么程序会产生这样的结果。实际上，通常首先你会看着一个黑屏（或窗口），奇怪于你的程序没有输出任何结果。对于 Windows 控制台程序，通常面临的第一个问题就是，在你能够看清楚输出结果之前（如果有的话），控制台窗口就消失了。一种解决办法是在 `main()` 的末尾调用 `std_lib_facilities.h` 中的 `keep_window_open()`。这样，窗口在退出前会等待一个输入。在给出一个输入关闭窗口之前，你就可以查看输出结果了。

在查找错误时，你要从程序中最后一个能够确认正确的语句开始，逐条语句仔细检查，就好像你就是计算机在执行这个程序。程序的输出是否与你的预计相同呢？当然是不会了，如果是的话，你就不用调试了。

- 通常，当你没有找到问题的时候，原因是你只“看到”了自己希望看到的，而不是你编写的程序，例如：

```
for (int i = 0; i <= max; ++i) {           // oops! (twice)
    for (int i = 0; i < max; ++i);         // print the elements of v
    cout << "v[" << i << "]==" << v[i] << '\n';
```

这个例子来自一个有丰富经验的程序员写的实际程序（我们认为它应该是在深夜编写的）。

- 通常，如果你没有找到问题所在，原因可能是在上一个你能确认的正确的输出和下一个输出（或没有输出）之间代码太多。大多数编程环境都提供逐条语句执行程序的功能（“单步执行”）。最终，你肯定能学会使用这些工具。但对于简单问题和简单程序来说，你可以通过在程序中临时加入一些额外的输出语句（利用 `cerr`）来帮助你检查运行结果。例如：

```
int my_fct(int a, double d)
{
    int res = 0;
    cerr << "my_fct(" << a << ", " << d << ") \n";
    // ... misbehaving code here ...
    cerr << "my_fct() returns " << res << '\n';
    return res;
}
```

- 在可能会隐藏错误的语句中，加入检查不变式（即永远成立的条件，参见 9.4.3 节）的语句，例如：

```
int my_complicated_function(int a, int b, int c)
// the arguments are positive and a < b < c
{
    if (!(0 < a && a < b && b < c)) // ! means "not" and && means "and"
        error("bad arguments for mcf");
    // ...
}
```

- 如果还是没有效果的话，在看起来不会有错的代码段中插入检查不变式的语句——如果你找不到错误，几乎可以肯定你是找错地方了。

陈述一个不变式的语句称为断言（`assertion` 或 `assert`）。

更有趣的是，还存在许多其他有效的编程方法。不同的人所使用的技术可能差别极大。调试

技术的差异中，有很多来自于要调试的程序的差异；另外一些则是源自于人们不同的思维方式。据我们所知，目前还没有一种最好的调试方法。但是有一件事要始终牢记：混乱的代码总是容易隐藏错误。尽你所能保证代码的简洁、有逻辑性和格式的工整吧，这将会大大减少你的调试时间。

5.10 前置条件和后置条件

现在，让我们回到前面的问题：如何处理函数的参数错误。基本上，函数调用是思考正确代码和捕捉错误的最佳位置：逻辑上，函数是一个独立计算的开始（函数返回是结束）。看看我们利用前一节介绍的调试技巧做了什么：

```
int my_complicated_function(int a, int b, int c)
// the arguments are positive and a < b < c
{
    if (!(0 < a && a < b && b < c))    // ! means "not" and && means "and"
        error("bad arguments for mcf");
    // ...
}
```

首先，我们（在注释中）说明了函数对于参数的要求，然后在函数开始处检查这一要求是否满足（如果不满足则抛出一个异常）。

这是一个很好的基本策略。函数对于它的参数的要求称为前置条件（pre-condition）：如果函数正确执行的话，这一条件必须为真。现在的问题是如果前置条件被违反的话（为假），那么我们应该怎么做。我们可以有两个选择：

- 1) 忽略它（希望/假设调用者会使用正确的参数）。
- 2) 检查它（并以某种形式报告错误）。

我们可以使用参数类型机制，它能让编译器进行最简单的前置条件检查，并在编译时报告错误。例如：

```
int x = my_complicated_function(1, 2, "horsefeathers");
```

这里，编译器会检查出“第三个参数应是整型”的函数要求（“前置条件”）被违反了。基本上，我们在本节中要讨论的是如何处理编译器不能检查的函数要求/前置条件。

我们的建议是前置条件一定要在注释里说明（这样调用者可以知道函数的要求）。一个没有注释文档的函数会被认为能够处理每种可能的参数值。但是我们能够确信调用者会读这些注释并遵守其中的规则吗？有些时候我们不得不相信调用者会这样做，但我们通常还是要遵循“被调用者进行参数检查”这一原则，它可以解释为“让函数检查自己的前置条件”。在没有其他原因的情况下，我们要坚持做到这一点。不做前置条件检查的常见理由包括：

- 没人会使用错误参数。
- 做前置条件检查会使我的程序变慢。
- 检查工作太复杂了。

当我们知道“谁”将调用这个函数的时候，第一个理由是很合理的，但在实际编程工作中，这一条件很难满足。

第二个理由成立的情况远比人们通常认为的要少得多。大多数情况下，它应该作为“过早优化”的例子被摒弃掉。如果前置条件检查被证实真的是程序的负担的话，你当然可以把它去掉。但是它所保证的程序正确性就不能那么容易获得了，本来是它能捕获的一些错误，又需要你花费许多不眠之夜来查找了。

第三个原因是最严重的。很容易地(特别是对有经验的程序员来说)找到这种例子:前置条件检查所做的工作比执行函数本身还要多得多。一个例子就是字典查询操作:前置条件是字典是已排序的,而验证一个字典已排序的代价要远远高于单纯的查询操作。有时候,编写前置条件检查代码以及判定这部分代码是否正确是很困难的。但是,每当你编写函数的时候,你都应该考虑是否可以编写一个快速的前置条件检查代码。除非你有足够的理由,否则始终应该为函数编写前置条件检查代码。

编写前置条件(即使是以注释形式)还可以显著地提高你的程序质量:它能强迫你考虑函数的需求是什么。如果你不能用几行注释把它简单、准确地描述出来的话,那么你可能还没有真正地理解你要做什么。经验表明,编写前置条件注释和前置条件检查代码有助于避免许多设计上的错误。我们说过我们讨厌调试工作,使用前置条件将有助于避免设计错误和及早发现使用错误,例如:

```
int my_complicated_function(int a, int b, int c)
// the arguments are positive and a < b < c
{
    if (!(0 < a && a < b && b < c))    // ! means "not" and && means "and"
        error("bad arguments for mcf");
    // ...
}
```

与下面的简化版本相比,上面的程序将会节省你的时间。

```
int my_complicated_function(int a, int b, int c)
{
    // ...
}
```

5.10.1 后置条件

使用前置条件将有助于我们避免设计错误和及早发现使用错误。这种显式说明需求的思想能够被应用在其他方面吗?是的,你可以马上联想到:返回值!毕竟,我们一般都需要声明函数的返回值是什么。也就是说,如果一个函数返回一个值的话,我们总是要约定这个返回值是怎样的(否则的话调用者如何知道得到的是什么呢)。让我们再次看一下面积函数(参见 5.6.1 节):

```
// calculate area of a rectangle;
// throw a Bad_area exception in case of a bad argument
int area(int length, int width)
{
    if (length <= 0 || width <= 0) throw Bad_area();
    return length * width;
}
```

这个程序检查了前置条件,但是它没有在注释里面对此进行说明(对于这么一个小函数来说,这还是可以接受的),而且假定计算是正确的(这种简单计算应该也没问题)。但是,我们可以更明确一些:

```
int area(int length, int width)
// calculate area of a rectangle;
// pre-conditions: length and width are positive
// post-condition: returns a positive value that is the area
{
    if (length <= 0 || width <= 0) error("area() pre-condition");
    int a = length * width;
    if (a <= 0) error("area() post-condition");
    return a;
}
```

我们不可能对后置条件进行完全检查,但是我们至少可以检查其中一部分:返回值是否是正数。

试一试 尝试找出一组数据，它们能够满足当前版本面积函数的前置条件，但不满足后置条件。

前置和后置条件提供了基本的程序完整性检查。从这个角度看，它们与不变式(参见 9.4.3 节)、正确性(参见 4.2 节和 5.2 节)和测试(参见第 26 章)等概念紧密相关。

5.11 测试

如何知道我们应该什么时候停止调试呢？不错，在找到所有错误前，我们应该继续调试，或者尽力去做。如何知道我们已经找到了最后一个错误了呢？答案是没有办法。“最后一个错误”是程序员之间的笑话：这种东西是不存在的。对一个大程序来说，我们永远不会找到“最后一个错误”。因为在查找错误的同时，我们还要忙于按照新需求修改程序。

除了调试以外，我们还需要一种系统地查找错误的方法，这称为测试，我们将在 7.3 节、第 10 章和第 26 章中详细介绍相关内容。基本上，测试是以一个较大的、经过系统选择的数据集作为输入来执行一个程序，然后把相关的结果与期望值进行比较。基于一组给定输入的一次程序运行称为一个测试用例(test case)。实际程序可能会需要上百万个测试用例来进行测试。基本上，系统测试不可能靠人工输入一个个测试用例。在介绍过后续几章内容并掌握了必要的工具后，我们再正式讨论测试。在此期间，我们需要谨记的是找到错误才是好的，这才是进行测试需要秉承的态度。看看下面的内容：

态度 1：我比任何程序都聪明！我将击败这些@#\$\$%^代码！

态度 2：这部分代码我已经打磨了两周时间了。它是完美的！

你认为谁会找出更多的错误？当然，最好的情况是一个有经验的人带着一点“态度 1”，沉着、冷静、耐心地对程序中所有可能出错的地方进行系统地检查。好的测试人员的价值不亚于与他相同重量的黄金。

我们会尽量系统地选择测试用例，一般会包括正确和不正确的输入数据。7.3 节中会给出第一个示例。



简单练习

下面是 25 个代码片段，每一个都要被插入到这个“框架”中：

```
#include "std_lib_facilities.h"

int main()
try {
    <<your code here>>
    keep_window_open();
    return 0;
}
catch (exception& e) {
    cerr << "error: " << e.what() << '\n';
    keep_window_open();
    return 1;
}
catch (...) {
    cerr << "Oops: unknown exception!\n";
    keep_window_open();
    return 2;
}
```

每段代码都有 0 个或多个错误。你的任务是找出并排除每个程序中的错误。当你排除了所有错误后，程序

编译、运行并输出“Success!”。即使你认为已经找到了一个错误，你仍然需要输入(原始的、未修改的)程序并测试它，因为你可能猜错了，或者程序中的还有其他错误。这个练习的另一个目的是让你感受一下编译器对不同错误的反应是什么样的。你不需要输入上面的程序框架 25 次，用剪切、粘贴或类似的技术就可以了。不要通过删除一条语句来逃避问题，你应该试着通过修改、增加或删除一些字符来排除问题。

1. `Cout << "Success!\n";`
2. `cout << "Success!\n";`
3. `cout << "Success" << "\n"`
4. `cout << success << endl;`
5. `string res = 7; vector<int> v(10); v[5] = res; cout << "Success!\n";`
6. `vector<int> v(10); v[5] = 7; if (v[5] != 7) cout << "Success!\n";`
7. `if (cond) cout << "Success!\n"; else cout << "Fail!\n";`
8. `bool c = false; if (c) cout << "Success!\n"; else cout << "Fail!\n";`
9. `string s = "ape"; boo c = "fool"<s; if (c) cout << "Success!\n";`
10. `string s = "ape"; if (s=="fool") cout << "Success!\n";`
11. `string s = "ape"; if (s=="fool") cout < "Success!\n";`
12. `string s = "ape"; if (s+"fool") cout < "Success!\n";`
13. `vector<char> v(5); for (int i=0; i<=v.size(); ++i) ; cout << "Success!\n";`
14. `vector<char> v(5); for (int i=0; i<=v.size(); ++i) ; cout << "Success!\n";`
15. `string s = "Success!\n"; for (int i=0; i<6; ++i) cout << s[i];`
16. `if (true) then cout << "Success!\n"; else cout << "Fail!\n";`
17. `int x = 2000; char c = x; if (c==2000) cout << "Success!\n";`
18. `string s = "Success!\n"; for (int i=0; i<10; ++i) cout << s[i];`
19. `vector v(5); for (int i=0; i<=v.size(); ++i) ; cout << "Success!\n";`
20. `int i=0; int j = 9; while (i<10) ++j; if (j<i) cout << "Success!\n";`
21. `int x = 2; double d = 5/(x-2); if (d==2*x+0.5) cout << "Success!\n";`
22. `string<char> s = "Success!\n"; for (int i=0; i<=10; ++i) cout << s[i];`
23. `int i=0; while (i<10) ++j; if (j<i) cout << "Success!\n";`
24. `int x = 4; double d = 5/(x-2); if (d=2*x+0.5) cout << "Success!\n";`
25. `cin << "Success!\n";`



思考题

1. 举出 4 种主要错误类型并给出它们的简洁定义。
2. 在学生练习程序中，什么类型的错误我们可以忽略？
3. 每一个完整的程序应该能提供什么保证？
4. 举出 3 种可以减少程序错误，开发出符合要求的软件的方法。
5. 为什么我们会讨厌调试？
6. 什么是语法错误？给出 5 个例子。
7. 什么是类型错误？给出 5 个例子。
8. 什么是连接器错误？给出 3 个例子。
9. 什么是逻辑错误？给出 3 个例子。
10. 列出 4 种本章中讨论的可能导致程序错误的因素。
11. 你如何能知道一个结果是合理的？回答这类问题，你会用到什么技术？
12. 对比一下由函数调用者来处理运行时错误和由被调函数来处理运行时错误的异同。
13. 为什么说使用异常比返回一个“错误值”要好？

14. 你应该如何测试一个输入操作是否成功?
15. 描述一下抛出和捕获异常的过程。
16. 有一个名为 `v` 的向量, 为什么 `v[v.size()]` 会导致值范围错误? 这一调用会导致什么结果?
17. 描述一下前置条件和后置条件的定义; 并举个例子(不能用本章中的 `area()` 函数), 最好是一个带有循环的计算过程。
18. 什么时候你不会测试前置条件?
19. 什么时候你不会测试后置条件?
20. 调试程序时的单步执行是指什么?
21. 在调试程序时, 注释会有什么帮助?
22. 测试与调试有什么不同?

术语

参数错误	异常	需求	断言	不变式	运行时错误
捕获	连接时错误	语法错误	编译时错误	逻辑错误	测试
容器	后置条件	抛出	调试	前置条件	类型错误
错误	范围错误				

习题

1. 如果你还没有完成本章中的“试一试”, 请先完成相关练习。
2. 下面的程序是获得摄氏温度值并将其转化为绝对温度。但这些代码有很多错误, 找到这些错误, 指出并修改它们。

```
double ctok(double c)    // converts Celsius to Kelvin
{
    int k = c + 273.15;
    return int
}

int main()
{
    double c = 0;    // declare input variable
    cin >> d;        // retrieve temperature to input variable
    double k = ctok("c");    // convert temperature
    Cout << k << endl;    // print out temperature
}
```

3. 绝对零度是能够达到的最低温度, 即 -273.15°C 或 0 K 。即使上面的程序是正确的, 当输入一个低于这个值的温度时, 程序也应输出错误结果。检查一下, 当输入一个低于 -273.15°C 的数值时, 主程序是否产生错误。
4. 重做练习 3, 但这次把错误处理放在 `ctok()` 中。
5. 给这个程序增加一些功能, 使它也可以把绝对温度转化为摄氏温度。
6. 编写一个程序, 它可以实现绝对温度转化为华氏温度和华氏温度转换为绝对温度(公式见 4.3.3 节)。用估计的方法(参见 5.8 节)看看你的结果是否合理。
7. 一元二次方程的形式如下

$$a \cdot x^2 + b \cdot x + c = 0$$

解这个方程, 用到二次公式:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

这里面有一个问题: 如果 $b^2 - 4ac$ 小于零的话, 它将出错。编写一个可以解一元二次方程的程序。建立一个可以计算二次方程根的函数, 给定 a, b, c , 如果 $b^2 - 4ac$ 小于 0 就抛出一个异常。让程序的主函数调用这个函数, 如果有错误由主函数捕获异常。当程序发现方程没有实根的时候, 输出相应的信息。你如何

确定程序的结果是合理的？你能检验结果的正确性吗？

8. 编写一个程序，读取一个整数序列，并计算前 N 个整数之和。它首先询问 N 的值，然后读取 N 个值并存入一个 `vector` 中，再计算前 N 个值之和。例如：

请输入你希望求和的值的数量：

3

请输入一些整数(按 'l' 结束输入)：

12 23 13 24 15l

前3个数(12 23 13)之和是 48

9. 修改练习 6 的程序：如果结果不能表示为整型的话，输出一个错误信息。
10. 修改练习 8 的程序：使用 `double` 而不是 `int`。同时保证 `vector` 中临近的数值是不同的，并输出 `vector` 中的数值。
11. 编写程序，输出尽量长的斐波那契序列，也就是说，序列的开始是 1 1 2 3 5 8 13 21 34，下一个数是前两个数的和。找出 `int` 能允许的最大的斐波那契数。
12. 实现一个名为“公牛和母牛”(不知道是谁命名的)的竞猜程序。程序用一个向量存储 4 个 0 到 9 之间的整数，用户的任务是通过重复猜测找到这些数。例如，如果存储的是 1234 而用户猜测的是 1359，程序的反馈结果是“一头公牛一头母牛”。因为用户的猜测中有一位正确(1)并且它在正确的位置(一头公牛)，有一位正确(3)但它在错误的位置(一头母牛)。反复这一猜测过程，直到用户找到 4 头公牛，即找到这 4 个数字且都在正确的位置。
13. 上一题的程序有点乏味，因为答案硬编码到程序中了。编写一个新版本：用户可以重复玩这个游戏(不需要停止或重启游戏)，每一次游戏生成一组新的 4 个数的集合。你可以通过 4 次调用 `std::lib_facilities.h` 中的随机数发生器 `randint(10)` 来生成 4 个随机数。但你会发现，当你重复执行这个程序的时候，每一次程序都会给出 4 个相同的数。为了避免这一点，你可以要求用户输入一个数(可以是任意数字)，然后在调用 `randint(10)` 之前先调用 `srand(n)`，这里 n 是用户所输入的数，这个 n 称为种子，不同的种子会生产不同的随机数序列。
14. 从标准输入中读入(星期，数值)对，例如：
- 星期二 23 星期五 56 星期二 -3 星期四 99
- 将一个星期中每一天对应的所有数值存入一个 `vector<int>` 中。输出 7 个向量中的值。打印每个 `vector` 中的值的总和。忽略输入中的非法日期，例如 `Funday`，但是接受同义词，例如 `Mon` 和 `monday`。输出被拒绝数值的个数。

附言

你认为我们过分强调错误了吗？作为初学者我们可能会这么想。显然，一个自然的反应是“这么简单的程序不可能出错”。但是，错误总是会发生的。许多世界上最聪明的人都惊讶和困惑于编写正确程序所具有的难度。根据我们的经验，好的数学家是最可能低估错误问题的人群。但我们都会认识到：所编写的程序一次通过是一件超出我们能力范围的事。你已经被警告过了！尽管我们会尽自己所能，但是错误不可避免地会出现在刚刚编写完的程序中。幸运的是，经过了 50 年或更长的时间，我们已经拥有了许多如何组织代码来最小化错误数目的经验，以及相关的错误查找技术。本章中介绍的技术和示例就是一个好的开端。

第6章 编写一个程序

“程序设计就是问题理解。”

——Kristen Nygaard

编写程序需要不断地细化所实现的功能及其表达方式。接下来的这两章详细讲述了程序的开发过程，从一个并不十分清晰的想法开始，经过分析、设计、实现、测试、再设计和再实现等步骤，最终实现预期目标程序。本章主要讨论程序结构、用户定义类型和输入处理等内容，目的是帮助读者了解在编写代码过程中如何去思考。

6.1 一个问题

程序的编写往往都是从一个问题出发，也就是说，借助程序来解决一个实际问题，因此正确理解问题对程序实现是非常关键的。毕竟，解决一个理解错误的问题的程序很可能是没有用处的，即使它是一个完美的程序。或许这个程序恰好对从来没有想到的某些问题是有用的，但这种幸运事件发生的概率非常小。因此，所设计的程序应该简单、清晰地解决要处理的问题。

在这个阶段，一个好的程序应该具有以下几个特点：

- 阐明设计和编程技术。
- 易于探究程序员做出的各种各样的决策及其相关考虑。
- 不需要很多新的语言结构。
- 对设计的考虑足够全面。
- 考虑所有可能的解。
- 解决一个易于理解的问题。
- 解决一个有价值的问题。
- 具有一个足够小，从而可完整实现、彻底理解的求解方案。

我们编写一个简单的计算器，实现计算机对输入表达式的常规算术运算。无疑这类程序是很有用的，在每个台式计算机中都安装有这样的计算器程序，甚至我们可以购买专门运行该程序的计算机：袖珍计算器。例如输入：

2+3.1*4

程序应该输出：

14.4

不幸的是，这样的计算器程序在我们的电脑上已经随处可见，它不会给我们带来任何新功能，但作为第一个程序，我们可以从其中获得很多内容。

6.2 对问题的思考

我们如何开始？大体上说，我们要做的就是对问题和问题求解方法进行思考。首先，考虑程序应该完成什么，人机交互的方式是怎样的。然后，考虑如何设计程序才能实现这样的功能。试着写出每个解决方案的简单框架，并检验它们的正确性。或许与朋友讨论这个问题及其解决方法，试着给朋友讲述你的想法，是一种很好的发现错误的方式，甚至比写下来都要好很多，因为

纸(或者计算机)不能对你的假设提出疑问,不能反驳你的错误观点。总之,设计不是一个孤独的过程。

不幸的是,不存在对所有人和问题都有效的通用解决策略。有些书通篇都在帮助读者学习更好地求解问题,其他大部分书籍则侧重于程序设计。我们并不那么做,相反,本书针对一些个人能够处理的小规模问题给出若干有价值的通用求解策略,然后以微型计算器程序为例对这些策略进行验证。

建议读者带着疑问阅读关于计算器程序的讨论。实际上,一个程序的开发要经过一系列版本,每个版本实现了我们得到的一些推论。很明显,某些推论是不完全的甚至是错误的,否则可以更早就结束本章的学习。随着讨论的深入,我们逐步给出了各种各样的关注点和推论的实例,这些都是设计者和程序员一直要面对和处理的,直到第7章结尾才完成这个程序的最终版本。

学习本章和第7章时要记住,实现程序最终版本的过程——提出部分解、产生想法和发现错误的历程——与程序最终版本本身同样重要,甚至比实现过程中碰到的语言技术细节更重要(后面还会讲到这些问题)。

6.2.1 程序设计的几个阶段

下面是程序开发涉及的几个术语。解决一个问题需要反复经历以下阶段:

- 分析(analysis): 判断应该做什么并且给出对当前问题理解的描述,称为需求集合(a set of requirements)或者规范(specification)。我们并不详细讨论如何开发和撰写这些规范,这已经超出本书的范围,但问题的规模越大,这种规范就越重要。
- 设计(design): 给出系统的整体结构图,并确定具体的实现内容以及它们之间的相互联系。作为设计的一个重要方面,要考虑哪些工具(如函数库)有助于实现程序的结构。
- 实现(implementation): 编写代码、调试和测试,确保程序完成预期的功能。

6.2.2 策略

下面是一些对很多程序设计项目都有帮助的建议:

- 要解决的问题是什么? 首要的事情是将要完成的目标具体化,包括建立问题的描述,或者分析已有描述的真实意图。这个时候应该站在用户而不是程序员的角度上,也就是说,应该考虑程序要实现什么功能,而不是怎样实现这些功能。例如,这个程序能够实现什么功能? 用户与程序以什么方式进行交互? 记住,大多数人都具有很丰富的计算机使用经验。
 - 问题定义清楚了吗? 事实上,我们无法十分清晰地定义一个现实问题,即使是一个学生练习题,也很难将其准确和具体地定义。但是,解决一个错误的问题是很遗憾的,所以必须弄清楚所要解决的问题是什么。另一个易犯的错误是我们容易把问题复杂化,在描述一个要处理的问题时总是表现得过于贪心/有野心。实际上,更好的方式是将问题简化,使程序易于定义、理解、使用 and 实现。一旦程序能够实现预期的功能,基于已有的经验可以实现它的第二版。
 - 看上去问题是可以处理的,但时间、技巧和工具是否足够? 从事一项不可能完成的项目是没有意义的。因此,如果没有足够的时间来实现(包括测试)一个程序,最好不要启动这个项目。否则,需要获取更多的资源(包括时间)或者修改需求来简化任务。
- 将程序划分为可分别处理的多个部分。为了解决一个实际问题,即使是一个最小的程序也能够进一步细分。
 - 你知道有哪些工具、函数库或者其他辅助手段吗? 答案是肯定的,因为即使在学习编程

的最初阶段,你也能使用C++标准函数库的部分内容。以后,还会慢慢学习如何使用标准函数库的更多功能,还会用到图形和GUI库、矩阵库等。在获得了一些编程经验之后,通过简单的网络搜索就能发现更多的函数库。记住,在编写真正实用的软件时,重新设计基本模块是没有价值的。在学习编程的时候是另外一回事,通过重新设计基本模块可以更清楚地了解其实现过程。通过使用函数库节约的时间可以用于解决问题的其他部分,或者休息。但是,如何知道一个函数库是否适合于目前的任务,或者程序性能是否满足要求是一个很困难的问题。一种解决方法是咨询同事、讨论组,或者在使用函数库前首先使用例子进行验证。

- 寻找可以独立描述的部分解决方案(或许能用在程序中的多个地方,甚至能用于其他程序)。发现这样的解决方案需要经验,因此本书提供了很多实例,目前我们已经用到了vector、string和iostream(cin和cout)。本章给出第一个完整的实例,展示了用户预定义类型(Token和Token_stream)的设计、实现和使用。第8章、第13至第15章给出了更多的实例,并给出了它们的设计思路。现在考虑一个类似的问题:如果要设计一辆汽车,首先应该确定它的每个组成部分,包括车轮、发动机、座椅、门把手等,在组装整车之前这些组件都可以独立生产。一辆汽车包含成千上万的组件,一个程序也是如此,只不过它的每个组件是代码而已。如果没有钢材、塑料和木材等原材料,我们是不能直接制造出汽车的,同样,没有语言提供的表达式、语句和类型,我们也不能直接编写出程序。设计和实现这种组件是本书的一个重要主题,也是软件开发的重要方法,参见第9章的用户自定义类型,第14章的类的分层结构和第20章的泛型。
 - 实现一个小的、有限的程序来解决问题的关键部分。当我们开始程序设计时,对要求解的问题并不十分了解。我们常常认为我们很了解(难道还不知道一个计算器程序是什么吗),但实际上并不是这样。只有充分思考(分析)并且实验(设计和实现)之后才能深入理解要求解的问题,才能编写出一个好的程序。因此,实现一个小的、有限的程序应做到以下两点:
 - 引出我们的理解、思想和工具中存在的问题。
 - 看看能否改变问题描述的一些细节使其更加容易处理。当我们分析问题并给出初步设计时,预先估计出所有问题几乎是不可能的。我们必须充分利用代码编写和测试过程中的反馈信息。
- 有时这样一个用于实验的小程序称为原型(prototype)。如果第一个程序不能工作或者很难在此基础上继续下去,可以将其丢弃,基于已有的经验重新设计一个原型程序,直到找到一个满意的版本为止。不要在一个混乱的版本上继续下去,否则将会越来越混乱。
- 实现一个完整的解决方案,最好是能够利用最初版本中的组件。理想情况是逐步构建组件来编写一个程序,而不是一下子写出所有代码。要不然,你就得希望奇迹发生了,期待一些未经检验的想法能够实现我们设想的功能。

6.3 回到计算器问题

如何与计算器进行交互?这个问题容易解决:因为我们知道如何使用cin和cout。但图形用户界面(GUI)直到第16章才讲述,因此这里使用键盘和控制台窗口。假设从键盘输入表达式,然

后计算并将结果显示在屏幕上。例如：

```
Expression: 2+2
Result: 4
Expression: 2+2*3
Result: 8
Expression: 2+3-25/5
Result: 0
```

$2+2$ 和 $2+2*3$ 等表达式是由用户输入的，而其他内容则是由程序输出的。我们选择输出“Expression:”提示用户输入表达式，当然可以使用“Please enter an expression followed by a newline”，但显得过于冗长，没有意义；另外，短提示符“>”又显得过于模糊。像这样尽早给出如何使用程序的例子是很重要的，这为程序最低限度应该实现哪些功能提出了非常实际的定义。在程序的设计与分析过程中，这样的实例通常称为用例 (use cases)。

当第一次碰到计算器问题时，大多数人对于程序的主要逻辑提出如下想法：

```
read_a_line
calculate      // do the work
write_result
```

像上面这样的描述称为伪代码 (pseudo code)，并不是真正的程序代码。在设计初始阶段，当对问题的定义并不完全清晰的时候，往往采用这种伪代码形式加以描述。例如，在上面的伪代码描述中，“calculate”是一个函数调用吗？如果是，它的参数是什么？在这个阶段回答这些问题还为时过早。

6.3.1 第一步尝试

在这个阶段，我们并没有准备好编写计算器程序。我们对问题还没有深入思考，不过思考总是比较困难的，而且像大多数程序员一样，我们急于编写程序代码。下面让我们试一试，编写一个简单的计算器程序，看看它将我们引向哪里。按照最初想法设计的程序如下：

```
#include "std_lib_facilities.h"

int main()
{
    cout << "Please enter expression (we can handle + and -): ";
    int lval = 0;
    int rval;
    char op;
    int res;
    cin >> lval >> op >> rval;    // read something like 1 + 3

    if (op == '+')
        res = lval + rval;    // addition
    else if (op == '-')
        res = lval - rval;    // subtraction

    cout << "Result: " << res << '\n';
    keep_window_open();
    return 0;
}
```

上面的程序读取运算符隔开的两个运算对象 (如 $2+2$)，计算并打印结果值 (本例为 4)，其中运算符左边的变量名为 lval，右边的变量名为 rval。

这个程序能够运行了！但如果这个程序不完整将会怎样？让程序运转起来感觉是很棒的！也许程序设计和计算机科学并不像大家所说的那么难。好吧，也许是这样的，但不要过早沉迷于这小小的成功。继续下面几项工作：

- 1) 进一步清理代码。
- 2) 加入乘法和除法(如 $2 * 3$)。
- 3) 加入处理多个操作符的功能(如 $1 + 2 + 3$)。

特别地,我们应该检查输入的内容是否符合要求(但由于匆忙,我们“忘记了”)。另外,如果一个变量的值可能是多个常量之一,检测它的值最好采用 switch 语句而不是 if 语句。

对于“ $1 + 2 + 3 + 4$ ”这种包含多个运算符的表达式,按照它们的输入顺序进行加法运算,也就是说,从 1 开始,输入 +2 后计算 $1 + 2$ (得到中间结果 3),输入 +3 后将其加到中间结果上,直到运算结束。经过一些简单的语法和逻辑修改之后得到如下程序:

```
#include "std_lib_facilities.h"

int main()
{
    cout << "Please enter expression (we can handle +, -, *, and /)\n";
    cout << "add an x to end expression (e. g., 1+2*3);";
    int lval = 0;
    int rval;
    char op;
    cin >> lval;      // read leftmost operand
    if (!cin) error("no first operand");
    while (cin >> op) { // read operator and right-hand operand repeatedly
        if (op != '+'') cin >> rval;
        if (!cin) error("no second operand");
        switch(op) {
            case '+':
                lval += rval; // add: lval = lval + rval
                break;
            case '-':
                lval -= rval; // subtract: lval = lval - rval
                break;
            case '*':
                lval *= rval; // multiply: lval = lval * rval
                break;
            case '/':
                lval /= rval; // divide: lval = lval / rval
                break;
            default:
                // not another operator: print result
                cout << "Result: " << lval << "\n";
                keep_window_open();
                return 0;
        }
    }
    error("bad expression");
}
```

程序没有错,但当我们输入 $1 + 2 * 3$ 时输出的结果是 9,而不是正确结果 7。同样,表达式 $1 - 2 * 3$ 的计算结果为 -3 而不是正确结果 -5。问题在于我们的计算顺序是错误的: $1 + 2 * 3$ 是按照 $(1 + 2) * 3$ 的顺序计算的,而不是通常的 $1 + (2 * 3)$ 。同样, $1 - 2 * 3$ 是按照 $(1 - 2) * 3$ 而不是 $1 - (2 * 3)$ 的顺序计算的。真糟糕!这种延续了几百年的人们所习惯的运算规则不会因为简化程序设计而消失,因此我们不能对“乘法的优先级高于加法”这种约定熟视无睹。

6.3.2 单词

我们必须“向前”看这一行表达式中有没有乘法或者除法运算符。如果有,必须调整这种简单的从左到右的计算顺序。然而,当我们试图这样做时,立刻遇到了很多困难:

1) 我们并没有必须要求表达式在一行输入, 例如:

```
1
+
2
```

目前的代码能够正确地计算其结果。

2) 如何在数字之间搜索“*” (或者“/”) 操作符, 而且该表达式有可能分散在多行?

3) 如何记住“*” 操作符的位置?

4) 如何不按从左到右的顺序计算表达式的值 (如 $1 + 2 * 3$)?

让我们做一回极端的乐观主义者, 首先解决前三个问题, 不去担心最后一个问题, 我们将在后面考虑它。

我们四处寻求帮助, 肯定有人知道如何从输入读取包括数字和操作符在内的表达式的方法, 而且以一种看起来非常合理的方式进行存储。答案就是“分词” (tokenize): 读取输入字符并组合为单词, 因此如果键入:

```
45+11.5/7
```

程序将产生一个单词列表

```
45
+
11.5
/
7
```

单词 (token) 是表示可以看做一个单元的一个字符序列, 例如数字或者运算符, 这也是 C++ 编译器处理源代码的方法。实际上, “分词” 在某种形式上是文本分析经常采用的方法。以 C++ 表达式为例, 可以看出所需的三种单词类型:

- 浮点常量: 如 C++ 定义的 3.14、0.274e2 和 42
- 运算符: +、-、*、/、%
- 括号: (,)

浮点常量看起来是个问题: 读取 12 比 12.3e-3 容易得多, 但计算器确实应该做浮点运算。同样, 我们的程序必须能识别括号, 否则计算器会显得没有用处。

如何在程序中表示这样的单词? 试图记录每个单词的开始 (和结束) 会非常繁琐、杂乱, 尤其是在允许表达式跨行的情况下。而且, 如果将数保存为字符串, 之后必须恢复它的数值。也就是说, 如果将数 42 存储为字符 4 和 2, 以后必须计算这些字符所表示的数值 42, 即 $4 * 10 + 2$ 。一种较好的解决方法是将每个单词表示为 (*kind*, *value*) 对, *kind* 表示单词是一个数、运算符还是括号。对于数 (在本例中, 也只有数), *value* 保存的就是它的数值。

那么, 如何在代码中表示一个 (*kind*, *value*) 对? 我们定义一个类型 Token 来表示单词。为什么要这么做? 记住我们为什么使用类型: 它们定义了所需要的数据以及对数据能执行的有效操作。例如, int 类型定义了整数及其加、减、乘、除和模等运算, 而 string 类型定义了字符串及其连接、下标等操作。C++ 语言及其标准函数库提供了很多类型, 如 char、int、double、string、vector 和 ostream 等, 但没有 Token 类型。事实上, 我们希望使用的类型成千上万甚至更多, 但语言及其标准函数库都未能提供, 其中比较有用的类型有第 24 章的 Matrix 类型、第 19 章的 Date 类型以及无限精度整数类型 (请尝试在互联网上搜索“Bignum”) 等。你将会意识到一种语言不可能提供成千上万的类型: 谁来定义和实现这些类型? 你怎样找到这么多类型? 参考手册将会多么厚? C++ 等高级语言通过让用户在需要的时候自己定义类型 (*user-defined type*) 而成功解决了这个问题。

6.3.3 实现单词

在程序中的单词应该是什么样的？换句话说，自定义的 Token 类型是什么样的？Token 必须能够表示运算符（如 +、-）和数值（如 42、3.14），即表示单词是什么类别以及保存单词的数值（如果有的话），如右图所示。

Token:		Token:	
kind:	plus	kind:	number
value:		value:	3.14

在 C++ 代码中有很多方式来表示这些类型，下面是最简单实用的一种方式：

```
class Token {           // a very simple user-defined type
public:
    char kind;
    double value;
};
```

Token 是一个类似于 int 或者 char 的类型，因此可用于定义变量及值。它有两个部分（称为成员）：kind 和 value。关键字 class 表示定义一个“用户自定义类型”，该类型可以有成员，也可以没有成员。第一个成员 kind 是一个 char 字符，因此可以方便地保存 '+' 和 '*' 表示加法和乘法。我们可以通过如下方式进行类型定义：

```
Token t;           // t is a Token
t.kind = '+';      // t represents a +
Token t2;          // t2 is another Token
t2.kind = '8';     // we use the digit 8 as the "kind" for numbers
t2.value = 3.14;
```

我们使用成员访问符号“object_name.member_name”访问成员，可以将 t.kind 读作“t 的 kind”，将 t2.value 读作“t2 的 value”。此外，可以像复制 int 型对象一样复制 Token 对象：

```
Token tt = t;      // copy initialization
if (tt.kind != t.kind) error("impossible!");
t = t2;            // assignment
cout << t.value;   // will print 3.14
```

给定 Token 类型，可以将表达式 $(1.5 + 4) * 11$ 表示为如下 7 个单词：

1	5	+	4	*	1	1
1.5			4			11

注意，像 '+' 这种简单的单词不需要值，因此我们不使用它的 value 成员。我们需要一个字符来表示单词“数”，由于 '8' 很明显不是一个运算符或者标点符号，这里选它标识单词“数”。使用 '8' 来表示“数”有点含混，但目前在这样用。

Token 是 C++ 用户自定义类型的一个实例。一个用户自定义类型可以有成员函数（操作）和数据成员，定义成员函数的理由有很多。这里仅提供两个成员函数，给出一种初始化 Token 的更有效方法：

```
class Token {
public:
    char kind;           // what kind of token
    double value;        // for numbers: a value
    Token(char ch)        // make a Token from a char
        : kind(ch), value(0) {}
    Token(char ch, double val) // make a Token from a char and a double
        : kind(ch), value(val) {}
};
```

这两个成员函数比较特殊，称为构造函数（constructor）。它们与类型具有相同的名称，用于初始化

(“构造”)Token 对象。例如：

```
Token t1('+'); // initialize t1 so that t1.kind = '+'
Token t2('8', 11.5); // initialize t2 so that t2.kind = '8' and t2.value = 11.5
```

在第一个构造函数中, `:kind(ch)`, `value(0)` 表示“将 `kind` 和 `value` 分别初始化为 `ch` 和 `0`”, 在第二个构造函数中, `:kind(ch)`, `value(val)` 表示“将 `kind` 和 `value` 分别初始化为 `ch` 和 `val`”。这两种情况都不需要其他操作来构造该 Token, 因此函数体为空: `{ }`。这种以冒号开始的特殊的初始化语法(成员初始化列表)仅用于构造函数中。

注意, 构造函数没有返回值, 不需要或者说不允许有返回类型。关于构造函数的更多介绍请参见 9.4.2 节和 9.7 节。

6.3.4 使用单词

现在, 或许我们能够实现完整的计算器程序了。然而, 可能前面的设计只有一小部分有用。在计算器程序中如何使用 Token? 我们可以将输入读到 Token 向量中:

```
Token get_token(); // function to read a token from cin
```

```
vector<Token> tok; // we'll put the tokens here
```

```
int main()
{
    while (cin) {
        Token t = get_token();
        tok.push_back(t);
    }
    // ...
}
```

接下来, 我们可以先读取一个表达式, 然后对其进行运算。例如, 对于 $11 * 12$ 可以得到右图。我们可以从中找到乘法符号及其操作数, 因此非常容易地执行乘法运算, 因为数字 11 和 12 是作为数字而不是字符串存储的。

8	*	8
11		12

接下来看一个更复杂的表达式, 给定 $1 + 2 * 3$, tok 包含 5 个 Token:

8	1	8	+	8
1		2		3

通过一个简单的循环就可以找到乘法操作符:

```
for (int i = 0; i < tok.size(); ++i) {
    if (tok[i].kind == '*') { // we found a multiply!
        double d = tok[i-1].value * tok[i+1].value;
        // now what?
    }
}
```

但接下来如何处理乘积 `d` 呢? 如何确定子表达式的计算顺序呢? 因为加法运算符在乘法运算符之前, 所以不能直接按照从左到右的顺序计算。我们可以试试从右到左计算! 但这么做对 $1 + 2 * 3$ 是正确的, 对 $1 * 2 + 3$ 又是错误的了。更糟的是 $1 + 2 * 3 + 4$, 必须“由内至外”计算: $1 + (2 * 3) + 4$ 。又出现了新的问题, 我们如何处理括号呢, 最终我们到底应该如何做呢? 似乎是走进了死胡同。我们必须后退一步, 先停止程序的编写, 重新考虑如何读取和分析输入表达式, 并计算它的值。

我们对于此问题求解(编写计算器程序)的第一次尝试以满怀热情开始, 但现在已经走到尽头了。对第一次编程来说, 这是很常见的。这不是灾难, 它使我们加深了对这个问题的理解。而且

在本例中，这次尝试还给我们带来了一个有用的概念：单词。我们今后会反复遇到(*name*, *value*) 对这种形式，单词是一个很好的实例。但是，我们必须确保这种轻率的、无计划的“编码”不会浪费太多时间。正确的做法是，在做过分析(理解问题)和设计(决定解决方案的整体结构)以后才进行程序设计。

试一试 另一方面，为什么不能找一个更简单的方法来解决这个问题？这看起来并不是那么困难。即便尝试没有什么效果，也可以增进我们对问题和最终求解方案的理解。现在马上思考你可以做些什么。比如对于表达式 $12.5 + 2$ ，我们可以先进行单词划分，然后确定表达式很简单，最终计算出结果。这个过程有一点凌乱，但它比较直接，或许我们可以沿着这个思路继续前进，找到很好的解决方法。接着考虑这种情况：在 $2 + 3 * 4$ 中既有 $+$ 又有 $*$ ，应该如何处理呢？仍然能够通过“蛮力”方式处理。但是，对于 $1 + 2 * 3 / 4 * 5 + (6 - 7 * (8))$ 这种更复杂的表达式又如何处理呢？如何处理像 $2 + * 3$ 和 $2 \& 3$ 这样的错误呢？稍微花些时间思考一下，或许可以在纸上写点什么，比如勾勒一下可能的解决方案，写出一些有趣或重要的输入表达式。

6.3.5 重新开始

现在再看一下问题，不要急于得出不完善的解决方案。必须注意，如果这个程序(计算器)运行后只计算一个表达式的值就结束，显然是不满足要求的。我们希望程序的一次执行能够对若干表达式进行计算，因此改进伪代码如下：

```
while (not_finished) {
    read_a_line
    calculate      // do the work
    write_result
}
```

很明显程序变得复杂了，但想想我们使用计算器的情景，你就会意识到一次做多个运算是很平常的事情。难道我们让用户做一次运算就启动一次程序吗？可以，但是在很多现代操作系统上启动程序的过程比较慢，因此最好不要这样做。

当我们审视上面的伪代码、我们最初的解决方案以及我们设计的使用实例，产生如下几个问题(其中某些给出了可能的答案)：

- 1) 如果输入表达式 $45 + 5/7$ ，那么如何找出其中的个体元素：45、 $+$ 、5、 $/$ 和7？(单词化!)
- 2) 如何标识表达式的结束？当然是用换行符！（要始终保持对“当然”的怀疑：“当然”不是一个有说服力的理由）。
- 3) 如何将表达式 $45 + 5/7$ 作为数据存储以便于计算？在计算加法之前必须先将字符4和5转换为整数45，即 $4 * 10 + 5$ 。（因此单词化是解决方案的一部分。）
- 4) 如何保证表达式 $45 + 5/7$ 的计算顺序为 $45 + (5/7)$ 而不是 $(45 + 5)/7$ ？
- 5) 表达式 $5/7$ 的值是多少？大约是0.71，但这不是一个整数。根据对计算器的使用经验可知，用户往往会期望得到一个浮点计算结果。我们应该允许输入表达式中出现浮点数吗？当然！
- 6) 可以使用变量吗？例如，我们能否使用下面的表达式

```
v=7
m=9
v*m
```

好主意，不过先放一放，我们还是先实现计算器的基本功能。

如何回答问题6可能是求解方案中最重要的抉择。在7.8节，你会看到如果我们决定实现变量功能，程序代码量将会是原来的两倍。让最初版本正常运行起来所花费的时间也将是原来的两

倍。以我们的估计，如果你是一个新手将会付出四倍的工作量，因而很可能最终放弃。在程序设计早期避免“功能蔓延”是很重要的，应该保证先构建一个简单的版本，只实现最基本的功能。一旦程序能够运转，你可以有更大的野心继续完善程序。分阶段实现一个程序比一次完成要简单得多。如果一开始就实现变量功能还有一个负面影响：很难抵挡进一步添加“漂亮特性”的诱惑。加上常用的数学函数如何？再加上循环功能怎么样？一旦开始便很难停下来。

从程序员的观点来看，问题1、3和4是令人困扰的。这几个问题相互关联，因为一旦找到一个45或者一个+，我们应该如何处理它们？也就是说，在程序中如何存储它们？很明显，单词化是整个解决方案的一部分，但仅仅是一部分而已。

一个有经验的程序员如何应对这些问题？当面对一个棘手的技术问题时，通常都有一个标准答案。我们知道，从计算机能够通过键盘接收符号输入开始，人们就已经开始编写计算器程序了。至少已经有50年的历史了，因此肯定有很成熟的解决方案。在这种情况下，有经验的程序员就会咨询同事、查阅文献。希望猛冲猛闯，一夜之间打破50年来的经验是很愚蠢的想法。

6.4 文法

对于如何理解表达式的含义，已经有标准的解决方法了：首先读入符号，将它们组合为单词。因此，如果键入

```
45+11.5/7
```

程序应该产生如下单词列表：

```
45
+
11.5
/
7
```

一个单词就是一个字符序列，用来表示一个基本单元，例如数字或者运算符。

在产生单词之后，我们的程序必须保证对整个表达式正确解析。例如，我们知道表达式 $45 + 11.5/7$ 的计算顺序是 $45 + (11.5/7)$ 而不是 $(45 + 11.5)/7$ ，但如何告诉程序这些有用的规则呢（例如，除法比加法优先级高）？标准的方法就是设计一个文法（grammar）来定义表达式的语法，然后在程序中实现这些文法规则。例如：

```
// a simple expression grammar:
```

```
Expression:
```

```
Term
```

```
Expression "+" Term    // addition
```

```
Expression "-" Term    // subtraction
```

```
Term:
```

```
Primary
```

```
Term "*" Primary       // multiplication
```

```
Term "/" Primary       // division
```

```
Term "%" Primary       // remainder (modulo)
```

```
Primary:
```

```
Number
```

```
"(" Expression ")"    // grouping
```

```
Number:
```

```
floating-point-literal
```

这是一个简单的规则集合，最后一条规则读作“一个 Number 是一个浮点常量”，倒数第二条规则表明“一个 Primary 是一个 Number，或者是 '(' 后接一个 Expression 再接一个 ')'”。针对 Expression 与 Term 的规则类似，都是依赖其后的规则来定义。

如 6.3.2 节, 我们从 C++ 定义中借用了如下几类单词:

- 浮点常量: 与 C++ 定义相同, 如 3.14、0.274e2 和 42 等
- 运算符: +、-、*、/、% 等
- 括号: (、)

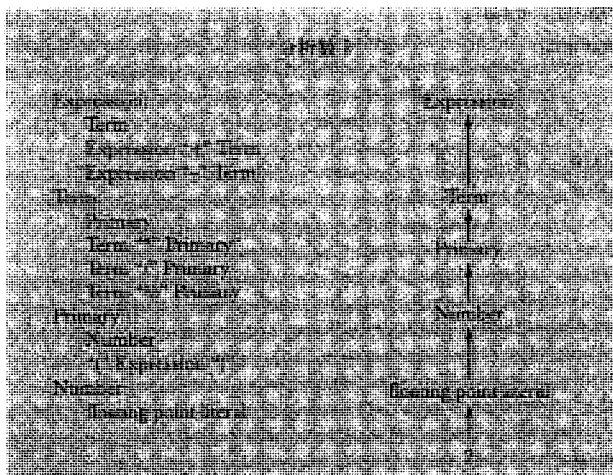
从最初的伪代码到现在使用单词和文法的方法, 在概念上是一个巨大的飞跃。这正是我们所期望的那种飞跃, 但不依靠帮助——前人的经验、参考文献和导师的指导是办不到的。

乍看起来, 文法好像根本没有意义, 语法符号看起来也是如此。然而, 请记住它是一种通用的、优美的符号(最终你会体会到这种符号的好处), 实际上, 你在中学时期或更早就有能力使用这样一套符号系统。你自己来计算 $1 - 2 * 3$ 、 $1 + 2 - 3$ 和 $3 * 2 + 4 / 2$ 这样的表达式, 显然是没有问题的, 如何计算已经深深印在你的头脑中了。但你能解释你是如何做的吗? 你能给一个从未学过传统算术的人解释清楚吗? 你的方法能用来计算任意运算符和运算数组组合出的表达式吗? 为了能清楚地表达计算方法, 而且足够详细、足够精确, 能被计算机所理解, 我们需要一种符号系统——对此, 文法是一种常规的、强有力的工具。

如何来读入一个文法呢? 基本方法是这样的: 对于给定输入, 从“顶层规则”Expression 开始, 搜索与输入单词匹配的规则。根据文法读取单词流的方式称为语法分析(parse), 实现该功能的程序称为分析器(parser)或者语法分析器(syntax analyzer)。语法分析器从左到右读取单词, 与我们输入和阅读的顺序一致。下面给出一个非常简单的实例: 2 是一个表达式吗?

- 1) 一个 Expression 必须是一个 Term 或者以 Term 结尾, 一个 Term 必须是一个 Primary 或者以 Primary 结尾, 一个 Primary 必须以 (或者 Number 开头。很明显, 2 虽然不是 (, 但它是一个浮点常量型 Number, 因此它是一个 Primary。
- 2) Primary (Number 2) 前没有 /、* 或者 %, 因此它是一个完整的 Term, 而不是 /、* 或者 % 表达式的结尾。
- 3) Term (Primary 2) 前没有 + 或者 -, 因此它是一个完整的 Expression, 而不是 + 或者 - 表达式的结尾。

因此, 根据文法, 2 是一个表达式, 分析步骤如下所示:

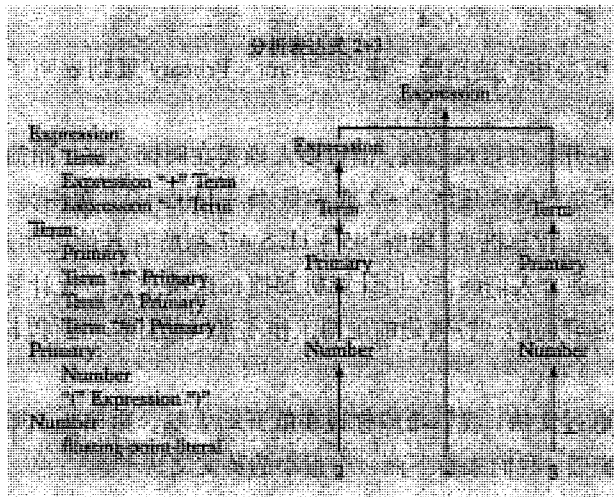


上面给出了根据定义解析表达式的方法, 由解析路径可知 2 是一个表达式, 因为 2 是一个浮点常量, 而一个浮点常量是一个 Number, 一个 Number 是一个 Primary, 一个 Primary 是一个 Term, 一个 Term 是一个 Expression。

下面给出一个更复杂的实例： $2 + 3$ 是一个 Expression 吗？很明显，许多推导过程与分析 2 时一样：

- 1) 一个 Expression 必须是 Term 或者以 Term 结尾，Term 必须是 Primary 或者以 Primary 结尾，Primary 必须以（开头或者是 Number。显然 2 不是左括号，而是一个浮点常量类型的 Number，因而是一个 Primary。
- 2) Primary (Number 2) 前面没有 /、* 或者 %，因此它是一个完整的 Term，而不是 /、* 或者 % 表达式的结尾。
- 3) Term (Primary 2) 后面跟着 + 运算符，因此它是 Expression 第一部分的结尾，必须寻找 + 运算符后面的 Term。与推导 2 是一个 Term 的方式一样，3 也是一个 Term。由于 3 后面没有 + 或 - 操作符，因此它是一个完整的 Term，而不是加/减表达式的一部分。因此， $2 + 3$ 符合 Expression + Term 规则，因此是一个 Expression。

我们还是以图形方式说明这个推导过程，为简化省略了浮点常量到 Number 规则的推导。

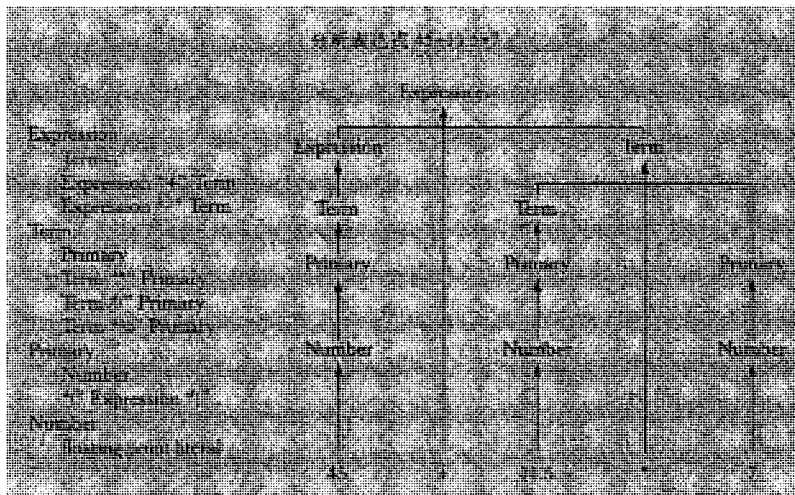


上图给出了根据定义分析表达式的路径，由分析路径可知 $2 + 3$ 是一个 Expression，因为 2 是一个 Term 类型的 Expression，3 是一个 Term，而 Expression 后接一个 + 再接一个 Term 构成一个 Expression。

我们对文法感兴趣的真正原因在于它可以帮助我们正确地分析同时包含 + 和 * 的表达式，下面看看如何处理 $45 + 11.5 * 7$ 。然而，计算机利用上述规则分析此表达式的详细过程是令人乏味的，因此我们省略其中一些熟悉的中间步骤，如分析 2 和 $2 + 3$ 的过程。很明显，45、11.5 和 7 都是浮点常量，因而都是 Number，也都是 Primary，因此我们忽略了所有 Primary 之下的规则。于是有：

- 1) 45 是一个 Expression，后面紧跟一个 +，因此需要寻找一个 Term，以便实现 Expression + Term 文法规则。
- 2) 11.5 是一个 Term，后面紧跟一个 *，因此需要找到一个 Primary，匹配 Term * Primary 文法规则。
- 3) 7 是一个 Primary，由 Term * Primary 规则可知 $11.5 * 7$ 是一个 Term。同样，根据 Expression + Term 规则可知， $45 + 11.5 * 7$ 是一个 Expression。特别地，这个表达式需要先算 $11.5 * 7$ ，然后再运算 $45 + 11.5 * 7$ ，正像我们所写的 $45 + (11.5 * 7)$ 。

下面给出推导过程的图示(省略了浮点常量到 Number 规则的推导):



上面给出了根据定义分析表达式的路径。注意, Term * Primary 规则表明了 11.5 先与 7 做乘法, 而不是与 45 做加法运算。

你会发现这种方法在开始时很难理解, 但很多人确实在阅读文法, 而简单的文法理解起来并不困难。然而, 我们并不是想教你理解 $2 + 2$ 或者是 $45 + 11.5 * 7$ 。很明显, 你已经知道这些了。我们只是在尝试找到一种让计算机来“理解” $45 + 11.5 * 7$ 和所有其他更加复杂的表达式的方法。实际上, 复杂的文法并不适合人们阅读, 但计算机却擅长这项工作。让计算机快速、准确地遵循这些规则进行分析是非常容易的。按精确的规则工作本来就是计算机所擅长的。

6.4.1 英文文法

如果你以前从未接触过文法, 我们希望你现在就开动大脑。事实上, 即使你以前曾经接触过文法, 现在还是要开动脑筋, 我们来看下面一个很小的英文文法子集:

Sentence :

Noun Verb // e.g., C++ rules

Sentence Conjunction Sentence // e.g., Birds fly but fish swim

Conjunction :

"and"

"or"

"but"

Noun :

"birds"

"fish"

"C++"

Verb :

"rules"

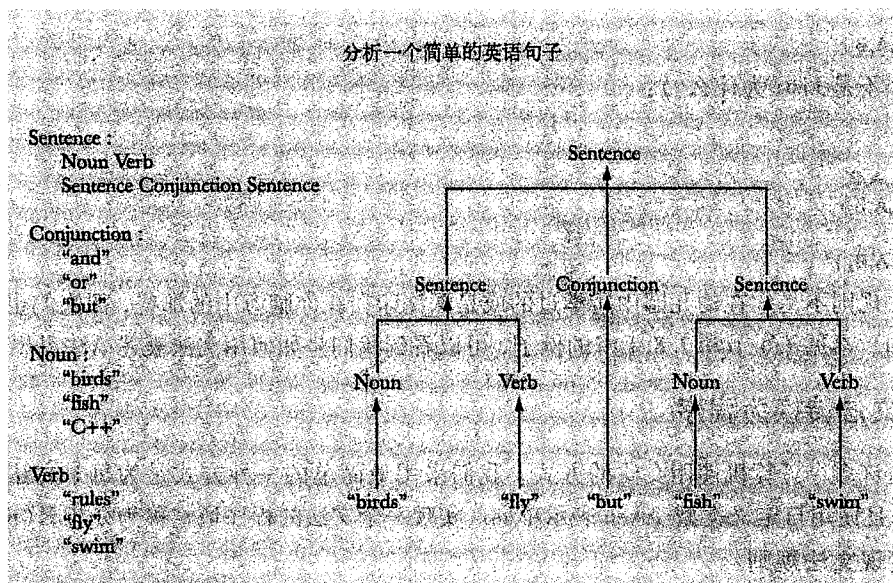
"fly"

"swim"

一个句子由语言的基本单元(例如名词、动词和连词)构成。根据文法规则可以分析一个句子, 确定哪些字是名词, 哪些是动词等。这个简单的文法也会产生一些没有意义的句子, 例如, “C++ fly and birds rules”, 但如何修正这一问题已超过本书的范围。

许多人已经在中学或者外语课(例如英语课)上学习过这些文法规则了, 这些文法规则是非常基本的。实际上, 这些规则深深地印在我们的脑海之中, 这也是神经学所研究的重要课题。

下面来看一下语法分析树，前面我们用它来分析表达式，这里用来描述简单的英语：



这些语法看起来并不是那么复杂。如果你不理解 6.4 节的内容，你现在可以回去头去重新阅读，经过第二次阅读你将会发现很多有用的东西。

6.4.2 设计一个文法

我们是如何设计出这些表达式的文法规则呢？“经验”是最诚实的回答，我们的方法不过就是人们所习惯的书写表达式文法的方法。然而，写一个简单的文法还是有一些相当直接的方法的：我们需要知道

- 1) 如何区分文法规则和单词。
- 2) 如何来排列文法规则(顺序)。
- 3) 如何表达可选的模式(多选)。
- 4) 如何表达重复的模式(重复)。
- 5) 如何识别出文法规则的开始。

不同的教材与不同的分析程序使用不同的符号约定和不同的术语。例如，一些人习惯称单词为终结符(terminal)，称规则为非终结符(non-terminal)或者产生式(production)。我们简单地将单词放在(双)引号中与规则相区分。而文法第一条规则为默认的起始规则。一个规则的可选模式放在不同行中。例如：

```

List:
    "{" Sequence "}"
Sequence:
    _Element
    Element " ," Sequence
Element:
    "A"
    "B"
  
```

因此上面文法的含义是，一个 Sequence 是一个 Element，或者是一个 Element 后面紧跟一个 Sequence，并且两者以逗号隔开。一个 Element 是字母 A 或是字母 B。List 是在花括号中的一个 Sequence。我们可以生成如下一些 List(如何生成的?)：

```
{ A }
{ B }
{ A, B }
{ A, A, A, A, B }
```

但下面这些不是 List(为什么?):

```
{ }
A
{ A, A, A, A, B }
{ A, A, C, A, B }
{ A B C }
{ A, A, A, A, B, }
```

上面的文法规则不是你在幼儿园中所学过的或是深深印在你脑海中的那些,但它们也并非什么高深的学问。参见 7.4 节和 7.8.1 节的例子,可以看到我们是如何用文法表述语法思想的。

6.5 将文法转换为程序

现在有许多令计算机使用文法的方式。我们采用最简单的一种方式是为每个文法规则写一个函数,并且使用自定义类型 Token 表示单词。实现一个文法的程序通常称为分析器(parser)。

6.5.1 实现文法规则

我们在计算器程序的设计中使用了四个函数,一个函数用于读入单词,其他三个函数分别实现文法的三条规则:

```
get_token()    // read characters and compose tokens
               // uses cin
expression()   // deal with + and -
               // calls term() and get_token()
term()         // deal with *, /, and %
               // calls primary() and get_token()
primary()      // deal with numbers and parentheses
               // calls expression() and get_token()
```

注意,每个函数只处理表达式的一个特定部分,将其他工作留给其他函数,这样可以简化函数的实现。如同每个人处理自己所擅长的问题,将其他不熟悉的问题留给同事完成一样。

这些函数应该具体做什么呢?每个函数应该根据其所实现的文法规则,调用其他文法函数,并利用 get_token() 获得规则所需要的单词。例如,当 primary() 函数实现(Expression)规则时,它必须调用

```
get_token()    // to deal with ( and )
expression()   // to deal with Expression
```

这些语法分析函数的返回值应该是什么呢?这个问题实际等价于——我们想得到什么结果呢?例如,对于 $2 + 3$, expression() 应该返回 5。毕竟,所有信息都包含其中了。这就是我们应该做的!这样做实际上回答了前面列表中最复杂的问题:“如何表示 $45 + 5/7$ 才有利于计算它的值?”我们在读入表达式时就计算它的值,而不是把它的某种表示形式存储到内存中。这个小小的想法其实是一个重要的突破!我们如果让 expression() 返回某种复杂的形式,随后再进行计算的话,程序规模会是直接计算值的版本的 4 倍。直接计算表达式的值会节省我们 80% 左右的工作量。

剩下的那个函数是 get_token(): 由于它只处理单词,而不是表达式,因此它不能返回一个子表达式的值。例如, + 和 (不是表达式。因此,它必须返回一个 Token 对象。因此有:

```
// functions to match the grammar rules:
Token get_token()    // read characters and compose tokens
double expression()  // deal with + and -
double term()        // deal with *, /, and %
double primary()     // deal with numbers and parentheses
```

6.5.2 表达式

下面首先编写 `expression()`，它的文法规则如下：

Expression:

Term

Expression '+' Term

Expression '-' Term

由于这是我们第一次尝试将一组文法规则转换为代码，将会经历一些不成功的开始。这是学习一种新技术常见的过程，我们可以从中学到很多有用的东西。特别地，通过观察相似代码段表现出令人吃惊的不同行为，初学者可以学到很多。阅读代码是积累编程技巧的有效途径。

6.5.2.1 表达式：第一次尝试

首先看一下 `Expression '+' Term` 规则，我们首先调用 `expression()`，然后寻找 `+`（和 `-`），最后是调用 `term()`：

```
double expression()
{
    double left = expression(); // read and evaluate an Expression
    Token t = get_token();      // get the next token
    switch (t.kind) {           // see which kind of token it is
        case '+':
            return left + term(); // read and evaluate a Term,
                                // then do an add
        case '-':
            return left - term(); // read and evaluate a Term,
                                // then do a subtraction
        default:
            return left;         // return the value of the Expression
    }
}
```

这个程序看起来不错。它几乎是文法的一个简单誊写，其结构确实非常简单：首先读入一个 `Expression`，然后判断它后面是否跟着一个“+”或者一个“-”，如果确是这样，则再读取 `Term`。

不幸的是，这里存在很大问题。怎样才能知道一个表达式的结尾在何处，以便于寻找一个“+”或者一个“-”呢？请记住，我们的程序从左到右读取输入，它不能预取符号来查看前面是否有“+”运算符。事实上，这个 `expression()` 函数只能执行到第一行代码，因为 `expression()` 在一直不停地调用自己，这种情况称为无限递归（infinite recursion）。实际上递归调用还是会停止的，因为每次调用都会消耗一定的内存空间，当计算机内存被耗光时，程序就会退出。递归的含义就是程序调用自身，并不是所有的递归都是无限的，递归是一种非常有用的程序设计技术（参见 8.5.8 节）。

6.5.2.2 表达式：第二次尝试

既然如此，我们能做什么呢？每一个 `Term` 都是一个 `Expression`，但一个 `Expression` 未必是 `Term`。也就是说，我们可以从寻找一个 `Term` 开始，但只有在找到一个“+”或者一个“-”时才算寻找一个完整的 `Expression`。例如：

```
double expression()
{
    double left = term(); // read and evaluate a Term
    Token t = get_token(); // get the next token
    switch (t.kind) {     // see which kind of token that is
        case '+':
            return left + expression(); // read and evaluate an Expression,
                                // then do an add
        case '-':
```

```

    return left - expression(); // read and evaluate an Expression,
                                // then do a subtraction
    default:
        return left;           // return the value of the Term
    }
}

```

实际上, 这个函数或多或少是可以正确运行的。我们在最终的程序中测试过它, 它完全可以分析我们输入的每一个合法的表达式, 甚至还能正确计算大多数表达式的值。例如, 对于 $1 + 2$, 首先读入一个 Term (值为 1), 接着是一个 $+$ 运算符, 然后是一个 Expression (恰好是一个值为 2 的 Term), 最后计算出结果 3。同样, $1 + 2 + 3$ 的计算结果为 6。我们还可以举出很多它可以正确运算的例子, 但我们还是简洁些: 对于 $1 - 2 - 3$, 会得到什么结果呢? 这个 `expression()` 函数会把 1 作为一个 Term 读入, 接着读入表达式 $2 - 3$ (由 Term 2 和后面 Expression 3 构成), 然后用 1 减去 $2 - 3$ 的值。换句话说, 它计算的是 $1 - (2 - 3)$ 的值, 其计算结果为 2 (正数 2)。但是, 我们在小学甚至更早就学过 $1 - 2 - 3$ 等价于 $(1 - 2) - 3$, 其结果是 -4 (负数 4)。

因此, 我们得到的是一个看似正确却不能得到正确结果的程序, 这是很危险的。这个例子尤其危险的地方在于, 它能够在很多情况下给出正确的结果。例如, 它能计算出 $1 + 2 + 3$ 正确的结果 6, 因为 $1 + (2 + 3)$ 等价于 $(1 + 2) + 3$ 。重要的是, 从程序设计的角度来看, 我们做错了什么吗? 每当发现错误时, 我们都应该问自己这个问题。这样可以帮助我们避免一而再, 再而三地犯同样的错误。

最基本的做法是阅读代码并猜测错误在哪里, 但这通常不是一个好方法。因为我们必须理解程序代码在做什么, 必须能解释它为什么对有些表达式计算正确, 对有些表达式则计算错误。

错误分析通常也是能找出正确求解方案的最好方法。在本例中, 我们定义 `expression()` 函数如下: 先读入一个 Term, 接着判断它后面是否有一个 $+$ 或者一个 $-$, 若有则寻找一个 Expression。实际上这实现了一个略微不同的文法:

```

Expression:
    Term
    Term '+' Expression    // addition
    Term '-' Expression    // subtraction

```

这与我们希望实现的语法的不同之处在于, 对 $1 - 2 - 3$ 这样的表达式, 我们希望解释为 Expression $1 - 2$ 后接 $-$ 再接 Term 3, 但得到的却是 Term 1 后接 $-$ 再接 Expression $2 - 3$; 也就是说, 我们希望 $1 - 2 - 3$ 意味着 $(1 - 2) - 3$, 但得到的却是 $1 - (2 - 3)$ 。

是的, 调试可能是一项非常乏味、棘手的工作, 也非常耗时。但在此例中, 我们确实是在使用在小学就学过的、可以帮我们避免很多错误的运算规则。潜在的困难——可能出错的地方在于我们必须把这些规则教给计算机, 但计算机却不善于学习这些内容。

注意, 我们实际上可以将 $1 - 2 - 3$ 定义为 $1 - (2 - 3)$ 而不是 $(1 - 2) - 3$, 那我们就不必再讨论这个问题了。但这显然是不行的, 我们不能改变人们习惯的算术运算规则。这就是困难所在: 常见的程序设计难题, 多数是因为程序必须符合传统的规则, 而这些规则早在计算机出现之前就已经建立起来并被人们所使用了。

6.5.2.3 表达式: 幸运的第三次

那么现在应该做什么呢? 再次回顾一下文法 (6.5.2 节中那个正确文法): 任何一个 Expression 都以一个 Term 开始, Term 后面可以跟一个 $+$ 或者一个 $-$ 。因此, 我们必须先寻找 Term, 看它后面是否有一个 $+$ 或者一个 $-$, 并且重复此步骤直到 Term 后面没有加号或者减号为止。例如:

```
double expression()
{
    double left = term();           // read and evaluate a Term
    Token t = get_token();         // get the next token
    while ( t.kind=='+' || t.kind=='-' ) { // look for a + or a -
        if (t.kind == '+')
            left += term();         // evaluate Term and add
        else
            left -= term();         // evaluate Term and subtract
        t = get_token();
    }
    return left;                   // finally: no more + or -; return the answer
}
```

这个程序可能有点混乱：我们必须引入一个循环来寻找加号与减号。而且，这里还有一些重复工作：对 + 和 - 的测试进行了两次，get_token() 函数也被调用了两次。这样导致程序的逻辑变得有点混乱，下面我们修改程序，去掉对 + 和 - 的重复测试。

```
double expression()
{
    double left = term();           // read and evaluate a Term
    Token t = get_token();         // get the next token
    while(true) {
        switch(t.kind) {
            case '+':
                left += term();     // evaluate Term and add
                t = get_token();
                break;
            case '-':
                left -= term();     // evaluate Term and subtract
                t = get_token();
                break;
            default:
                return left;       // finally: no more + or -; return the answer
        }
    }
}
```

注意，除了循环，该程序与第一次尝试的程序非常相似(参见 6.5.2.1 节)。我们所做的就是用循环语句替代了 expression() 中对自身的调用。换句话说，我们把 Expression 文法规则中的 Expression 转换为循环语句，在循环语句中寻找后接 + 和 - 的 Term。

6.5.3 项

Term 的文法规则与 Expression 规则非常相似：

```
Term:
    Primary
    Term '*' Primary
    Term '/' Primary
    Term '%' Primary
```

因此，它们的代码基本相同。下面是第一次尝试：

```
double term()
{
    double left = primary();
    Token t = get_token();
    while(true) {
        switch (t.kind) {
            case '*':
                left *= primary();
```

```

        t = get_token();
        break;
    case '/':
        left /= primary();
        t = get_token();
        break;
    case '%':
        left %= primary();
        t = get_token();
        break;
    default:
        return left;
    }
}
}

```

不幸的是，程序没有编译成功：编译器给出了错误信息——C++ 对浮点数没有定义模运算(%)。当我们回答前面列出的第五个小时问题时“我们应该允许输入表达式中出现浮点数吗？”，我们做出了肯定的回答“当然！”，实际上我们当时并没有全面考虑这个问题，从而陷入了功能蔓延的困境。这种情况经常会发生！那么我们应该如何处理呢？我们可以在运行时检查运算符%的两个运算数是否为整数，若不是则给出错误信息；或者简单地将操作符%排除在外，本书中就选择这种简单方法。我们可以随时将运算符%加进来(参见7.5节)。

在去掉运算符%以后，函数能够正常运行了，能够正确分析 Term 并进行计算。然而，有经验的程序员会注意到 term() 中存在一个不可接受的情况。如果我们输入 2/0 会发生什么情况？C++ 程序中零不能作为除数，否则计算机硬件会检测出这一情况，并终止程序，给出一些无用的错误信息。一个新手很难发现问题在哪里，所以，最好在程序中检查这种情况并给出一个恰当的错误提示：

```

double term()
{
    double left = primary();
    Token t = get_token();
    while(true) {
        switch (t.kind) {
            case '*':
                left *= primary();
                t = get_token();
                break;
            case '/':
                {
                    double d = primary();
                    if (d == 0) error("divide by zero");
                    left /= d;
                    t = get_token();
                    break;
                }
            default:
                return left;
        }
    }
}

```

为什么我们把处理/的语句放入一个语句块内呢？这是编译器规定的，如果要在 switch 语句中定义和初始化变量，必须把它们放在一个语句块内。

6.5.4 基本表达式

基本表达式的文法规则也很简单：


```

Primary:
    Number
    '(' Expression ')'

```

它的实现代码有点混乱，因为其中有很多可能导致语法错误的地方：

```

double primary()
{
    Token t = get_token();
    switch (t.kind) {
    case '(': // handle '(' expression ')'
        {
            double d = expression();
            t = get_token();
            if (t.kind != ')') error("'')' expected");
            return d;
        }
    case '8': // we use '8' to represent a number
        return t.value; // return the number's value
    default:
        error("primary expected");
    }
}

```

基本上，与 `expression()` 和 `term()` 函数相比并没有什么新内容。我们使用了相同的语言指令、相同的单词处理方式以及相同的编程技巧。

6.6 试验第一个版本

为了执行这些计算器函数，需要实现 `get_token()` 函数并提供一个 `main()` 函数。`main()` 函数比较简单，仅仅用于 `expression()` 函数的调用和结果输出。

```

int main()
try {
    while (cin)
        cout << expression() << '\n';
    keep_window_open();
}
catch (exception& e) {
    cerr << e.what() << endl;
    keep_window_open ();
    return 1;
}
catch (...) {
    cerr << "exception \n";
    keep_window_open ();
    return 2;
}

```

错误处理部分还是老样式（参见 5.6.3 节）。我们把 `get_token()` 函数的实现留到 6.8 节介绍，这里只是用它来测试计算器程序的第一个版本。

试一试 计算器程序的第一个版本（包括 `get_token()`）在文件 `calculator00.cpp` 中。

请尝试编译、运行它，并验证结果。

不出所料，计算器程序的第一个版本并没有很好地按我们期望的方式来工作。于是我们不禁要问“它为什么不按我们期望的方式工作呢？”或者更进一步，“它为什么像这样工作呢？”以及“它能做什么呢？”我们输入数字 2 并换行，程序没有反应。再敲一个换行来看看程序是否进入睡眠状态了，仍然没有反应。接着输入数字 3 并换行，程序还是没有反应！再输入数字 4 接着换行，程序终于给出一个应答 2！此时屏幕显示如下：

2

3

4

2

接着继续输入 5 + 6，程序输出 5，此时屏幕显示如下：

2

3

4

2

5+6

5

除非你以前有过编程经验，否则你多半会陷入深深的迷惑之中！事实上，即使是一个有经验的程序员对此也可能感到迷惑。接下来该如何处理呢？这时你应该尝试结束程序。但应该如何结束程序呢？我们没有在程序中设置结束命令，但一个错误可以导致程序结束运行。因此，你可以输入一个 x，程序会输出 Bad token 然后结束运行。终于有这么一次，程序能按我们的设想工作了！

但是，我们忘记将屏幕上的输入与输出信息加以区分了。在解决主要问题之前，让我们先对输出做些改动，易于呈现出程序做了什么事情。我们在输出内容前增加一个 =，将其与输入信息区分开来：

```
while (cin) cout << "=" << expression() << '\n'; // version 1
```

现在，重新输入与上一次运行完全一样的符号，我们得到如下结果：

2

3

4

= 2

5+6

= 5

x

Bad token

很奇怪！我们试着理解程序做了些什么。我们还尝试了另外几个例子，但还是集中精力看看这个例子，下面几点令人迷惑不解：

第一次输入 2、3 并换行以后，为什么程序没有反应呢？

在输入 4 以后，为什么程序输出的是 2 而不是 4 呢？

在输入 5 + 6 以后，为什么程序回答的是 5 而不是 11 呢？

产生这些奇怪的结果有很多可能的原因，其中一些将在第 7 章详细讨论，这里我们只是简单思考。程序会产生算术运算错误吗？这几乎是不可能的。但确实结果是错误的：输入 4 的结果不应该是 2，5 + 6 的结果是 11 而不应该是 5。我们再试着输入 1 2 3 4 + 5 6 + 7 8 + 9 10 11 12 然后换行，看看会出现什么结果。我们将得到：

1 2 3 4+5 6+7 8+9 10 11 12

= 1

= 4

= 6

= 8

= 10

哈！没有输出 2 或者 3，而且为什么输出 4 而不是 9(4 + 5)呢？为什么输出 6 而不是 13(6 + 7)呢？仔细观察：程序在每三个单词中输出一个！是不是程序“吃掉”了一些字符而没有让它们参加运算呢？确实是这样，考虑 expression() 函数：

```

double expression()
{
    double left = term();           // read and evaluate a Term
    Token t = get_token();         // get the next token
    while(true) {
        switch(t.kind) {
            case '+':
                left += term();      // evaluate Term and add
                t = get_token();
                break;
            case '-':
                left -= term();      // evaluate Term and subtract
                t = get_token();
                break;
            default:
                return left;        // finally: no more + or -; return the answer
        }
    }
}

```

当 `get_token()` 返回的单词不是 '+' 或者 '-' 时, 我们简单地从 `expression()` 函数返回了。我们没有使用那个单词, 也没有把它保存下来用于后面的计算。这是不明智的做法, 甚至没有判断单词是什么就把它丢弃的做法不是一个好主意。快速查看一下可以发现, `term()` 函数中也存在同样的问题。这就解释了我们的计算器为什么会每处理一个单词后就会“吃掉”后面的两个。

我们来修改 `expression()` 函数, 使其不再“吃掉”单词。那么当程序不需要下一个单词(`t`)时, 应该把它放在哪儿呢? 我们可以给出很多复杂精巧的方案, 但在此选择最显而易见的一种(你一看这个方法就会知道它确实“显而易见”): 如果其他某个函数需要使用该单词, 而此函数从输入流读入单词的话, 我们将该单词放回输入流, 它就能够再次被此函数读取! 实际上, 我们是可以把字符退回标准输入流 `istream` 中的, 但那不是我们真正想要的。我们希望的是处理输入的单词, 而不是把输入流变得混乱。我们需要的是一个专门用于单词处理的输入流, 能够将已经读出的单词重新放回去。

假设我们已经有一个称为 `ts` 的单词流“`Token_stream`”, 并假设 `Token_stream` 的成员函数 `get()` 用于返回下一个单词, 成员函数 `putback(t)` 用于将单词 `t` 放回单词流。一旦了解了如何使用单词流之后, 我们将在 6.8 节实现 `Token_stream`。给定了 `Token_stream`, 我们可以重写 `expression()` 函数, 令其将不使用的单词放回 `Token_stream`。

```

double expression()
{
    double left = term();           // read and evaluate a Term
    Token t = ts.get();             // get the next Token from the Token stream

    while(true) {
        switch(t.kind) {
            case '+':
                left += term();      // evaluate Term and add
                t = ts.get();
                break;
            case '-':
                left -= term();      // evaluate Term and subtract
                t = ts.get();
                break;
            default:
                ts.putback(t);      // put t back into the token stream
        }
    }
}

```

```

        return left;    // finally: no more + or -; return the answer
    }
}

```

另外, 必须对 `term()` 函数做同样的修改:

```

double term()
{
    double left = primary();
    Token t = ts.get();    // get the next Token from the Token stream

    while(true) {
        switch (t.kind) {
            case '*':
                left *= primary();
                t = ts.get();
                break;
            case '/':
                {
                    double d = primary();
                    if (d == 0) error("divide by zero");
                    left /= d;
                    t = ts.get();
                    break;
                }
            default:
                ts.putback(t);    // put t back into the Token stream
                return left;
        }
    }
}

```

对最后一个分析函数 `primary()`, 只需要把 `get_token()` 函数改为 `ts.get()`, `primary()` 函数使用它读入的每一个单词。

6.7 试验第二个版本

现在, 我们准备测试程序的第二个版本。输入 2 并换行以后, 程序没有输出, 再次换行程序仍然没有输出。输入 3 并换行以后程序输出 2, 输入 2 + 2 并换行以后程序输出结果 3。此时屏幕上显示:

```

2
3
=2
2+2
=3

```

由此可知, 也许在 `expression()` 和 `term()` 中使用 `putback()` 并没有解决问题。下面做另一个测试:

```

2 3 4 2+3 2*3
=2
=3
=4
=5

```

程序给出了正确的结果! 但最后一个结果(6)却不见了。这里仍然存在一个单词预读方面的问题。但是, 这次的问题不是我们的程序“吃掉”了字符, 而是它不能返回表达式的运算结果, 除非再输入后续表达式。也就是说, 一个表达式的结果不是被立即输出, 而被推迟到程序读入下一个

表达式的第一个单词以后才输出。不幸的是，只有在输入下一个表达式并回车以后，程序才能读到那个单词。程序本身没有错误，只是它的输出有些延迟。

如何改进这个问题？一个很明显的方法是加入一个“输出命令”。我们使用分号标识一个表达式的结束并触发结果的输出。另外，我们再增加一个“退出命令”，实现程序的正常退出。字符 `q` (表示“quit”) 用于表示退出命令是很恰当的。在原来版本的 `main()` 函数中，有

```
while (cin) cout << "=" << expression() << "\n"; // version 1
```

我们把它改成下面这样，可能有点复杂，但却更加实用：

```
double val = 0;
while (cin) {
    Token t = ts.get();

    if (t.kind == 'q') break; // 'q' for "quit"
    if (t.kind == ';')        // ';' for "print now"
        cout << "=" << val << "\n";
    else
        ts.putback(t);
    val = expression();
}
```

现在的计算器程序真正可用了。例如：

```
2;
= 2
2+3;
= 5
3+4*5;
= 23
q
```

现在，我们有了一个比较好的计算器程序的初步版本。虽然还不是我们最终想要的那样，但是可以把它作为进一步完善的基础。重要的是，现在的版本可以正常运行，然后就可以逐步改进问题、增加功能，并在这个过程中一直保持一个能正常运行的版本。

6.8 单词流

在改进计算器程序之前，我们先给出 `Token_stream` 的实现。毕竟，程序在没有获得正确输入之前是不能正确运行的。因此，我们首先实现 `Token_stream`，但并不是想偏离计算器程序这个主题太远，只是首先完成一个尽量小的可用程序。

计算器程序的输入是一个单词序列，如 $(1.5 + 4) * 11$ (参见 6.3.3 节)。我们需要从标准输入 `cin` 中读入字符，并且能够向程序提供运行时需要的下一个单词。另外，我们发现计算器程序经常多次读入一个单词，因此应该把它们保存起来便于后续使用。这是最典型也是最基本的功能，当严格从左到右读入 $1.5 + 4$ 时，在没有读入 $+$ 之前，你如何判断浮点数 1.5 已经完整读入了呢？实际上，在遇到 $+$ 之前，我们完全有可能是在读入 1.55555 而不是 1.5 的过程中。因此，我们需要一个“流”，当我们需要一个单词时，可以调用 `get()` 函数从流中产生一个单词，并且可以利用 `putback()` 把单词放回流中。根据 C++ 的语法规则，我们必须先定义 `Token_stream` 类型。

你可能注意到了前面 `Token` 定义中的 `public:`，那里使用 `public` 并没有特别的原因。但对于 `Token_stream`，则必须使用 `public` 来限定相应的函数。C++ 用户自定义类型通常由两部分构成：公有接口 (用“`public:`”标识) 和具体实现 (用“`private:`”标识)。这样做主要是为了将用户接口 (用户方便使用类型所需的) 和具体实现 (实现类型所需的) 分开，希望没有对用户的理解造成困难：

```

class Token_stream {
public:
    // user interface
private:
    // implementation details
    // (not directly accessible to users of Token_stream)
};

```

显然，我们经常既扮演用户的角色，又扮演实现者的角色。但是，弄清楚用户使用的公有接口和仅由实现者使用的具体实现之间的区别，对于组织程序代码是非常重要的。公有接口应该只包含用户需要的内容，比如提供一组函数，包括初始化对象的构造函数在内。私有实现包括实现公有函数所必须的内容，包括用于处理复杂细节的数据和函数，而这些都是用户不必知道也不应该直接使用的。

下面详细给出 `Token_stream` 类型。用户需要 `Token_stream` 完成什么功能？很明显，需要 `get()` 和 `putback()` 两个函数，这也是我们设计单词流这个概念的原因。`Token_stream` 能够从标准输入读入字符，从中解析出单词，因此，类中应包含能创建 `Token_stream` 对象，并令它能从 `cin` 读入字符的函数。于是，最简单的 `Token_stream` 定义如下所示：

```

class Token_stream {
public:
    Token_stream();           // make a Token_stream that reads from cin
    Token get();              // get a Token
    void putback(Token t);    // put a Token back
private:
    // implementation details
};

```

这就是一个用户使用 `Token_stream` 所需要的全部内容。有经验的程序员可能会对 `cin` 是字符的唯一输入源感到惊讶，但这里我们决定只从键盘输入字符，第7章的一个练习将重新审视这个决定。

为什么我们使用了较长的名字 `putback()` 而不是 `put()` 呢？逻辑上看 `put()` 已经足够了。我们这样做是为了重点强调一下 `get()` 和 `putback()` 之间的不对称性，这是一个输入流，不包含能用来输出的函数。在 `istream` 中也实现了 `putback()` 函数：在系统中保持命名的一致性是比较重要的，有助于记忆和避免不必要的错误。

我们现在可以创建、使用 `Token_stream` 对象了：

```

Token_stream ts;           // a Token_stream called ts
Token t = ts.get();        // get next Token from ts
// ...
ts.putback(t);             // put the Token t back into ts

```

下面我们要做的就是实现计算器程序的剩余部分了。

6.8.1 实现 `Token_stream`

现在，我们实现 `Token_stream` 中的三个函数。如何表示一个 `Token_stream` 呢？也就是说，需要在 `Token_stream` 中存储什么数据才能完成相应的功能呢？放回 `Token_stream` 的任何单词都需要存储空间。但为了简单起见，这里规定每次只能放回一个单词，对我们的计算器程序（和其他许多类似的程序）来说这已经够用了。因此，我们只需要声明一个单词所需的存储空间和一个指示该存储空间占用或空的状态的指针。

```

class Token_stream {
public:
    Token_stream();           // make a Token_stream that reads from cin
    Token get();              // get a Token (get() is defined elsewhere)
    void putback(Token t);    // put a Token back
};

```

```
private:
    bool full;           // is there a Token in the buffer?
    Token buffer;        // here is where we keep a Token put back using putback()
};
```

现在，我们可以来定义（“编写”）三个成员函数了，首先定义比较简单的构造函数和 `putback()` 函数。

在构造函数中只需设置 `full`，指示缓冲区为空即可。

```
Token_stream::Token_stream()
    :full(false), buffer(0)    // no Token in buffer
{
}
```

当我们在类外定义一个成员时，必须指明这个成员属于哪个类，为此，需采用如下语法：

类名 :: 成员名

在前面的代码中，我们以这种方式定义了 `Token_stream` 的构造函数，构造函数是一个与类具有相同名字的成员函数。

为什么我们要在类的外部定义一个成员呢？主要是为了保持代码清晰：类的定义主要说明类能够做什么。成员函数定义则指明如何做，因此，我们倾向于将其放在“别的地方”，避免和类定义混在一起分散注意。我们的理想是，程序中的每个逻辑实体都很简短，能完整地显示在屏幕上的一页内。如果将成员函数定义放在别处，是能做到这点的，但如果将其放在类的定义中（“类内”成员函数定义），将很难满足这个要求。

我们使用成员初始化表来初始化类的成员（参见 6.3.3 节）：`full(false)` 将 `Token_stream` 的成员 `full` 设置为 `false`，`buffer(0)` 使用“哑单词”（dummy token）来初始化成员 `buffer`，哑单词是我们设计的专门用于此处的初始化的。6.3.3 节中 `Token` 的定义要求每个 `Token` 对象必须被初始化，因此不能忽略 `Token_stream::buffer`。

`putback()` 成员函数的功能是将其参数放回 `Token_stream` 的缓冲区中：

```
void Token_stream::putback(Token t)
{
    buffer = t;    // copy t to buffer
    full = true;   // buffer is now full
}
```

关键字 `void` 指出 `putback()` 函数不返回任何值。如果我们想确认不发生这种情况：连续两次调用 `putback()` 函数，期间没有用 `get()` 读取放回流的内容，可以增加一个测试：

```
void Token_stream::putback(Token t)
{
    if (full) error("putback() into a full buffer");
    buffer = t;    // copy t to buffer
    full = true;   // buffer is now full
}
```

对 `full` 的测试用来检查前置条件“缓冲区中没有单词”。

6.8.2 读单词

所有的读入操作都是 `get()` 函数完成的，如果在 `Token_stream::buffer` 中没有单词，`get()` 函数必须从 `cin` 读入字符并将它们组成单词：

```
Token Token_stream::get()
{
    if (full) {        // do we already have a Token ready?
        // remove Token from buffer
```



```

        full=false;
        return buffer;
    }

    char ch;
    cin >> ch;    // note that >> skips whitespace (space, newline, tab, etc.)

    switch (ch) {
    case ';':    // for "print"
    case 'q':    // for "quit"
    case '(': case ')': case '+': case '-': case '*': case '/':
        return Token(ch);    // let each character represent itself
    case '.':
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
    {
        cin.putback(ch);    // put digit back into the input stream
        double val;
        cin >> val;    // read a floating-point number
        return Token('8',val);    // let '8' represent "a number"
    }
    default:
        error("Bad token");
    }
}

```

下面我们详细分析一下 `get()` 函数。首先检测缓冲区中是否已经有单词了，如果有就直接返回该单词：

```

    if (full) {    // do we already have a Token ready?
        // remove Token from buffer
        full=false;
        return buffer;
    }

```

只有当 `full` 为 `false` 时(表明缓冲区中没有单词)，我们才需要处理输入字符。此时，我们逐个读入字符并进行适当的处理，在其中寻找括号、运算符和数字，遇到任何其他字符我们都将调用 `error()` 而结束程序：

```

    default:
        error("Bad token");

```

`error()` 函数在 5.6.3 节中描述过，我们将其声明包含在 `std_lib_facilities.h` 文件中。

我们必须考虑如何表示不同类型的单词，也就是说，必须为 `kind` 成员选择不同的值。为了简单起见，也为了易于调试，我们令一个单词的 `kind` 域就保存括号、运算符本身。这使得括号和运算符的处理异常简单：

```

    case '(': case ')': case '+': case '-': case '*': case '/':
        return Token(ch);    // let each character represent itself

```

坦率地讲，我们在第一个版本中忘记了处理表示打印的 ';' 和表示退出的 'q' 这两个符号，我们在第二个版本中将这部分代码添加进来。

6.8.3 读数值

现在，我们必须处理数值，事实上这不是一件容易的事。如何获得 123 这个数值呢？当然，它可由 $100 + 20 + 3$ 得来，但 12.34 又如何获得呢？另外，我们应该允许使用科学计数法(如 $12.34e5$)吗？为了正确实现这些功能，可能需要花几个小时甚至几天时间，幸运的是，我们可以不必做这个工作。输入流能够解析 C++ 文字常量，并能将其转换为 `double` 类型的数值。因此，我们所要做的只是如何在 `get()` 函数中告诉 `cin` 完成这些工作而已：

```

case '!':
case '0': case '1': case '2': case '3': case '4': case '5': case '6': case '7':
case '8': case '9':
    {      cin.putback(ch);          // put digit back into the input stream
      double val;
      cin >> val;                    // read a floating-point number
      return Token('8',val);        // let '8' represent "a number"
    }
}

```

在某种程度上，我们是随意选择了 '8' 来表示“数值”这类单词。

那么，我们如何知道输入中出现了一个数值呢？如果根据经验来推测，或者是参考 C++ 文献（如附录 A），我们会发现一个数值常量必须以一个阿拉伯数字或者小数点开头。因此，我们可以在程序中检测这些符号，来判断是否出现数值。接下来，我们希望 cin 完成数值的读取，但我们已经读入了第一个字符（一个阿拉伯数字或小数点）。因此，我们需要将第一个字符的数值和 cin 读入的后续字符的值结合起来。例如输入 123，我们会得到 1，cin 读入 23，我们需要将 100 与 23 相加。真是太繁琐了！幸运的是（并不是偶然的），cin 与 Token_stream 的工作方式类似，也可以把已经读出的字符放回输入流中。因此，不用做繁琐的数学运算，我们只需把第一个字符放回 cin，然后由 cin 读入整个数值。

请注意，我们如何一次又一次地避免做复杂的工作，代之以寻找简单的解决方案——通常是借助于 C++ 库。这就是程序设计的本质：不断寻找更简单的方法。这与“优秀的程序员都是懒惰的”（看起来有些好笑）不谋而合。从这个角度说（当然，也只有从这个角），我们应该“懒惰”，如果能找到一个更简单的方法，我们何必写那么多代码呢？

6.9 程序结构

有谚语说：只见树木不见森林。同样，如果我们只关心一个程序中的函数、类等，也会失去对程序的理解。因此，下面就忽略细节，看看程序的结构：

```

#include "std_lib_facilities.h"

class Token { /* ... */ };
class Token_stream { /* ... */ };

Token_stream::Token_stream() : full(false), buffer(0) { /* ... */ }
void Token_stream::putback(Token t) { /* ... */ }
Token Token_stream::get() { /* ... */ }

Token_stream ts;           // provides get() and putback()
double expression();        // declaration so that primary() can call expression()

double primary() { /* ... */ }    // deal with numbers and parentheses
double term() { /* ... */ }       // deal with *, /, and %
double expression() { /* ... */ } // deal with + and -

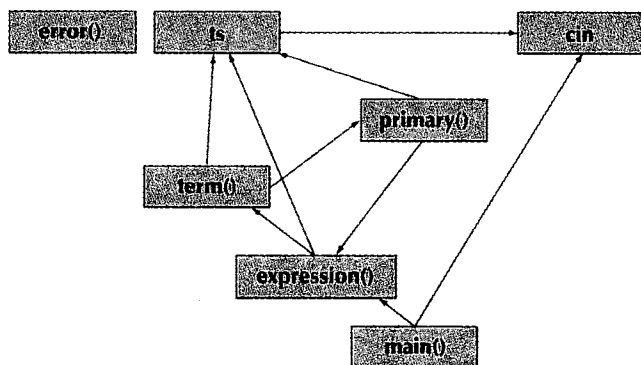
int main() { /* ... */ }         // main loop and deal with errors

```

在这里，声明的顺序是很重要的，变量在被声明之前是不能使用的，因此 ts 必须在 ts.get() 使用之前声明，error() 必须在分析函数之前声明。在调用图中有一个非常有趣的循环：expression() 调用 term()，term() 调用 primary()，primary() 又调用了 expression()。

下面是调用关系的图描述（由于所有的函数都调用 error()，因此将其省略）。

这就意味着我们不能简单地定义这三个函数：没有任何一种顺序能满足先定义后使用的原则。因此，至少有一个函数必须先只给出声明而非定义。我们选择先声明 expression() 函数，这种



方式称为前置声明 (forward declare)。

现在的计算器程序已经能正常工作了吗？在某种程度上确实可以了。它可以正常编译、运行、正确计算表达式，并给出适当的错误信息。但它是按照我们所希望的方式工作吗？答案并不出人意料——它并不能真正按我们的意图工作。6.6 节给出了程序的第一个版本并消除了一个严重的错误，6.7 节中的第二个版本并没有多少改进。但这没有关系，这是意料之中的。我们本来的目标就是写一个可以运行的程序，能用来验证我们的基本思路，从中获得反馈，对于这个目标来说现在的版本已经足够好了。从这个角度来说，它是成功的，但试一下，它仍然存在很多问题！

试一试 编译、运行上面设计的计算器程序，看看它能完成什么功能，并指出它为什么如此工作。

简单练习

本练习对一个有很多错误的程序进行一系列改进，使其变得更加有用。

1. 编译文件 `calculator02buggy.cpp` 中的计算器程序，你需要找到并修正一些错误，才能使程序编译通过，这些错误本书中并未涉及。
2. 把用于控制程序退出的命令符 `q` 换成 `x`。
3. 把用于控制程序输出的命令符 `;` 换成 `=`。
4. 在 `main()` 函数中增加一条欢迎信息：
**“Welcome to our simple calculator.
 Please enter expressions using floating-point numbers.”**
5. 改进欢迎信息，提示用户可以使用哪些运算符，以及如何输出结果和退出程序。
6. 找出并改正 `calculator02buggy.cpp` 计算器程序中存在的三个不太明显的逻辑错误，使计算器能够产生正确结果。

思考题

1. “程序设计就是问题理解”的含义是什么？
2. 本章详细讲述了计算器程序的设计，简要分析计算器程序应该实现哪些功能？
3. 如何把一个大问题分解成一系列易于处理的小问题？
4. 为什么编写一个程序时，先编写一个小的，功能可控的版本是一个好主意？
5. 为什么功能蔓延是不好的？
6. 软件开发的三个主要阶段是什么？
7. 什么是“用例”？
8. 测试的目的是什么？
9. 根据本章的描述，比较 `Term`、`Expression`、`Number` 和 `Primary` 的不同点。
10. 在本章中，输入表达式被分解为 `Term`、`Expression`、`Number` 和 `Primary` 等组成部分，试按这种方式分析表达式 $(17 + 4) / (5 - 1)$ 的构成。

11. 为什么程序中没有名为 `number()` 的函数?
12. 什么是单词?
13. 什么是文法? 语法规则是什么?
14. 什么是类? 类的作用是什么?
15. 什么是构造函数?
16. 在 `expression()` 函数中, 为什么 `switch` 语句的默认处理是退回单词?
17. 什么是“预读取”?
18. `putback()` 函数的功能是什么? 为什么说它是有用的?
19. 在 `term()` 函数中, 为什么难以实现取模运算符%?
20. `Token` 类的两个数据成员的作用是什么?
21. 为什么把类的成员分成 `private` 和 `public` 两种类型?
22. 对于 `Token_stream` 类, 当缓冲区中有一个单词时, 调用 `get()` 函数会发生什么情况?
23. 在 `Token_stream` 类的 `get()` 函数中, 为什么在 `switch` 语句中增加了对 ';' 和 'q' 的处理?
24. 应该从什么时候开始测试程序?
25. “用户自定义类型”是什么? 我们为什么需要这种机制?
26. 对于一个 C++ “用户自定义类型”, 其接口是什么?
27. 我们为什么要依赖代码库?

术语

分析	文法	函数原型	class	实现	伪代码
类成员	接口	public	数据成员	成员函数	语法分析
设计	语法分析器	单词	被0除	private	用例

习题

1. 如果你尚未做本章“试一试”中的练习, 现在就做一下。
2. 在程序中增加对 `{}` 的处理, 令其与 `()` 作用一致, 这样, `{(4+5)*6}/{(3+4)}` 就是一个合法的表达式。
3. 在程序中增加对阶乘运算符(用! 表示)的处理, 例如, 表达式 `7!` 表示 `7*6*5*4*3*2*1`。阶乘的优先级高于 `*` 和 `/`, 也就是说, `7*8!` 表示 `7*(8!)` 而不是 `(7*8)!`。通过修改文法来描述优先级更高的运算符, 为了与数学中阶乘的定义统一, 我们规定 `0!` 等于 1。
4. 定义包含一个字符串和一个值两个成员类 `Name_value`, 给出其构造函数(与 `Token` 的构造函数有些类似), 然后使用 `vector<Name_value>` 而不是两个 `vector` 重做第 4 章的习题 19。
5. 将 `the` 加到 6.4.1 节中的“英语”文法中, 以便能描述“The birds fly but the fish swim”这样的语句。
6. 根据 6.4.1 节中给出的“英语”文法, 编写程序判断一个句子是否符合英语语法。假设每个句子都以句号(.)结束, 句号的两边有空格。例如, “birds fly but the fish swim.” 是一个合法的句子, 但“birds fly but the fish swim”(缺少句号)和“birds fly but the fish swim.”(句号之前没有空格)都不是正确的句子。对输入的每个句子, 程序能够输出“OK”或者“not OK.”。提示: 不要为单词的处理而困扰, 直接使用 `>>` 来读入字符串即可。
7. 为逻辑表达式编写一个文法。逻辑表达式与算术表达式是非常相似的, 除了前者使用逻辑运算符!(非)、~(补)、&(与)、|(或)和^(异或), 而后者使用算术运算符。其中, ! 与 ~ 是前缀一元运算符, 异或运算的优先级高于或运算, 因此 `x|y^z` 表示 `x|(y^z)` 而不是 `(x|y)^z`; 与运算的优先级高于异或运算, 因此 `x^y&z` 表示 `x^(y&z)`。
8. 使用四个字母而不是四个数字重做第 5 章的习题 12 中的“Bulls and Cows”游戏。
9. 编写一个程序, 读入数字并将其组合为整数。例如, 读入字符 1、2、3 时得到整数 123, 程序应输出“123 is 1 hundred and 2 tens and 3 ones”。数值由一至四个数字构成, 以 `int` 类型输出。提示: 如何从字符 '5' 得到数值 5 呢? 可通过字符 '5' 减字符 '0' 得到, 也就是 `'5' - '0' == 5`。
10. 排列是一个集合的有序子集。例如, 你要从 60 个数中选取 3 个数排成金库密码, 则共有 $P(60, 3)$ 种排

列。其中函数 P 由如下公式定义

$$P(a, b) = \frac{a!}{(a-b)!}$$

其中 $!$ 是前缀阶乘运算符, 例如, $4!$ 表示 $4 * 3 * 2 * 1$ 。组合与排列相似, 差别在于不关心对象的顺序。例如, 如果你想从 5 种不同口味的冰淇淋中选出 3 种制作香蕉圣代, 那么你不必关心香草冰淇淋是先加入的还是后加入的, 因为无论如何香草冰淇淋都得加进去。组合的计算方式如下:

$$C(a, b) = \frac{P(a, b)}{b!}$$

设计一个程序, 让用户输入两个数字, 询问用户是计算排列还是组合, 然后输出计算结果。这项工作包含以下几个步骤: 首先分析上面给出的需求, 确定程序需要完成的功能; 然后进入设计阶段, 编写伪代码, 并将其分解为多个子模块。程序应该具有错误检查机制, 保证程序对错误的输入产生恰当的错误提示信息。

附言

了解输入的含义是程序设计的重要组成部分, 每个程序都会以某种形式面对这个问题。其中, 又以了解那些由人类直接生成的信息的含义最为困难。例如, 语音识别仍然是一个非常困难的研究课题。当然, 这类问题中也有一些较为简单, 例如本章所研究的计算器, 可以通过用文法描述输入的方式来处理。

第7章 完成一个程序

“不到最后，不见分晓。”

——歌剧谚语

编写程序需要不断地改进你要实现的功能及其表达方式。第6章中我们给出了一个能够正确运行的计算器程序的初步版本，本章将对其进行进一步的完善和优化。“完成程序”意味着使程序更易于用户使用，更方便开发者维护——包括改进用户接口、添加一些重要的错误处理机制、增加一些有用的功能、重构代码使之易于理解和修改。

7.1 介绍

当你的程序第一次正常运行时，你大约只完成了一半工作。如果是一个大程序或者一个由于不能正常运行而造成不良后果的程序，那连一半工作也没完成。一旦程序能初步正常运行，编程的真正乐趣就开始了！从这里开始，我们可以在初步版本上试验各种不同的想法。

本章将引导你如何以一名专业程序员的眼光来优化第6章给出的计算器程序。值得注意的是，本章提出的程序相关的问题以及一些思考，远比计算器程序本身有趣得多。本章将通过一个实例讲述如何在实际需求和约束条件的压力下逐步优化程序。

7.2 输入和输出

让我们回到第6章开始，你会发现我们当初决定采用提示信息“Expression:”提示用户输入表达式，用提示信息“Result:”提示输出计算结果。迫于使程序尽快运行起来的压力，我们忽略了这些看似不重要的细节。这是很常见的，我们不可能一开始就考虑所有情况。因此，当我们停下来反思的时候，发现忘记了最初想要实现的一些功能。

对于某些程序设计任务，原始需求是不能改变的。这一原则过于死板，会给问题求解方案的设计带来很多困难。假定我们可以修改需求，我们又该如何做呢，应该修改哪些需求，哪些又应该保持不变？我们真的让程序输出提示信息“Expression:”和“Result:”吗？仅仅靠“想”是不行的，最好是实际试验一下，看看哪种方式效果更好。现在我们输入

```
2+3; 5*7; 2+9;
```

输出结果为：

```
= 5  
= 35  
= 11
```

如果在程序中添加“Expression:”与“Result:”，将得到如下结果：

```
Expression: 2+3; 5*7; 2+9;  
Result : 5  
Expression: Result: 35  
Expression: Result: 11  
Expression:
```

我们相信，一些人会喜欢前一种风格，而其他人会喜欢后一种。因此可以考虑给予用户选择

自己喜欢的风格的权力。但对于这个简单的计算器程序来说,提供两种输入/输出风格供用户选择显得过于繁琐。因此,我们必须确定使用哪种风格。我们认为输出“Expression:”与“Result:”令程序有点复杂,而且会分散注意力。如果使用这些提示信息的话,真正有用的表达式输入与结果输出在屏幕显示窗口中只占据很少一部分,而表达式和结果才是我们真正关心的内容,其他内容不应分散注意力。另一方面,应该把用户输入的表达式和计算机输出的结果区分开,否则用户可能会无法分辨出结果。在最初调试程序时,我们用 = 表示计算结果的输出。类似地,我们也可以使用一个简短的“提示符”来提示用户输入——字符 > 经常作用用户输入提示符:

```
> 2+3;
= 5
> 5*7;
= 35
>
```

这种方式看起来好多了,我们只需对 main() 函数中的主循环做一点改动即可实现这种方式:

```
double val = 0;
while (cin) {
    cout << "> "; // print prompt

    Token t = ts.get();
    if (t.kind == 'q') break;
    if (t.kind == ';')
        cout << "=" << val << '\n'; // print result
    else
        ts.putback(t);
    val = expression();
}
```

不幸的是,如果在一行上输入多个表达式,其输出仍然比较混乱:

```
> 2+3; 5*7; 2+9;
= 5
> = 35
> = 11
>
```

根本原因是我们在开始开发程序时就认为用户不会在一行中输入多个表达式(至少我们假设用户不会这样)。对于这种情况,我们期望的输出方式如下:

```
> 2+3; 5*7; 2+9;
= 5
= 35
= 11
>
```

这种显示方式看起来很合理,但不幸的是,实现它却很麻烦。首先看一下 main() 函数,我们希望只有在后面不跟着符号 = 的情况下,才输出提示符 >, 这能够实现吗? 答案是否定的,因为我们根本没有办法判定这种情况! 因为程序是在 get() 函数调用之前输出提示符 > 的,而此时无法知道 get() 函数是真正读取了新字符,还是简单地将已经从键盘读入的字符组成单词返回给我们。换句话说,我们可能不得不弄乱 Token_stream 才能实现上述输出形式。

我们认为现在的输出形式已经基本满足要求了,因此不再进行改进。如果将来我们发现必须修改 Token_stream 类,那时再来重新考虑这个决定。不过,修改程序的主要数据结构,只是为了获得一点小小的改进,这是不明智的。而且,我们还未对计算器程序进行过全面的测试,因此目前我们决定不做输出形式上的改进。

7.3 错误处理

当你的程序能够初步运行时，你应该做的第一件事就是打破它——也就是给它各种输入，期望它表现出错误的行为。“期望”的意思是，在这个阶段，我们所面临的挑战是要发现尽可能多的程序错误，以便在最终交付用户之前将其修正。如果你做这项工作时的态度是“我的程序已经正常运行了，我是不会犯错误的！”那么你将不会发现很多错误，而一旦真的发现错误时，你又会非常沮丧。你应该调适自己的心理，进行程序测试时的正确态度应该是“我能打败它！我比任何程序都聪明，即使是我自己编写的程序！”我们可以使用一些正确的和不正确的表达式混合在一起的输入来测试计算器程序，例如：

```
1+2+3+4+5+6+7+8
1-2-3-4
!+2
;;;
(1+3;
(1+);
1*2/3%4+5-6;
();
1+;
+1
1++;
1/0
1/0;
1++2;
-2;
-2;;;
1234567890123456;
'a';
q
1+q
1+2; q
```

试一试 尝试用一些不同的“问题表达式”来测试计算器程序，看看你能使它表现出多少种不同的错误行为。你能使程序崩溃吗——使程序跳过错误处理机制而直接输出平台的出错信息？我们认为你做不到。你能让程序异常退出而不输出任何错误信息吗？我想你是可以做到的。

这种技术称为测试(test)。有些人专门从事这项工作，负责找出程序中的错误。测试是软件开发中很重要的一个环节，而且并非像想象的那样枯燥，实际上是可以很有趣的，我们将在第26章中详细讨论程序测试的一些细节问题。关于程序测试的一个重要问题是“能否对程序进行系统测试从而发现所有的错误？”这个问题没有一个普适的答案，也就是说，没有任何一个答案对所有程序都成立。不过，对于大多数程序而言，严格的测试通常都会获得很好的效果。程序测试最重要的环节之一是系统性地设计测试用例，为了防止测试设计不全面的情况，你可以用一些“不合理的”输入来测试程序。例如，对计算器程序输入：

```
Mary had a little lamb
srtvrqtiewcbet7rewaewre-wqcntrretewru754389652743nvcqnwq;
!@#$%^&*()~:;
```

在此，我再一次使用了我习惯性的做法：将电子邮件的内容（包括邮件头、邮件正文等所有内容）输入给编译器，来测试编译器的反应。这看起来不合情理，因为“没有人会这样做”。但在实际应用中，完美的程序应该能捕获所有错误，并且能够从“奇怪的输入”中快速恢复正常运行，而不是

只能处理那些“合乎情理”的输入。

在测试计算器程序时，第一个棘手的问题是当输入下列非法表达式时，程序窗口会立刻关闭：

```
+1;
0
!+2
```

稍加思考或者跟踪一下程序的执行过程就会发现，问题在于，在输出错误信息后，程序窗口就立刻关闭了。这是因为我们保持窗活跃是为了等待用户输入字符。然而，对于上述三种输入而言，程序在读入所有字符之前就检测到了一个错误，因此输入行中还剩下未被读入的字符。但是，程序不能区分它是“剩余字符”还是用户在看到提示信息“Enter a character to close window”后输入的字符。于是，这个“剩余字符”就被程序认为是关闭窗口的命令，导致程序窗口被关闭。

我们可以对 `main()` 函数稍做修改来处理这个问题（参见 5.6.3 节）：

```
catch (runtime_error& e) {
    cerr << e.what() << endl;
    // keep_window_open():
    cout << "Please enter the character ~ to close the window\n";
    char ch;
    while(cin >> ch)    // keep reading until we find a ~
        if (ch=='~') return 1;
    return 1;
}
```

基本上，我们将 `keep_window_open()` 函数完全替换为新的代码，来处理上述问题。但需要注意，如果 ~ 恰好是发生错误之后的下一个输入字符，上述问题仍然存在，不过这种情况出现的可能性就小得多了。

我们可以编写一个新版本的 `keep_window_open()` 函数处理这个问题，它接受一个字符串参数，只有用户在看到提示信息后输入这个字符串，程序才会关闭窗口，其简单实现如下：

```
catch (runtime_error& e) {
    cerr << e.what() << endl;
    keep_window_open("~~");
    return 1;
}
```

此时输入如下内容：

```
+1
!1~~
0
```

计算器程序会在输出错误信息后给出如下提示信息，

```
Please enter ~~ to exit
```

直到用户输入 ~~ 后才退出。

计算器程序从键盘获取输入，这使得程序测试过程非常乏味：每当对程序做出改动，我们都要（再一次）从键盘手工输入许多测试用例，以测试程序修改是否正确。因此，最好能将测试用例保存起来，用一个单独的命令就能输入这些用例进行测试。对于以 UNIX 为代表的操作系统来说，在不改变程序的情况下，令 `cin` 从文件而不是键盘输入数据，令输出到 `cout` 的内容转向到文件，是非常容易的。但如果你所使用的操作系统中，这种输入/输出重定向很难实现，则必须对程序代码进行修改（参见第 10 章）。

现在考虑下面两个输入：

```
1+2; q
```

和

```
1+2 q
```

我们希望程序对于这两个输入都能够在输出结果(3)之后退出。但奇怪的是, $1 + 2q$ 确实是这样的, 而看起来显然更正确的 $1 + 2;q$ 却引发了一个“Primary expected”错误。我们应该如何来查找这个错误呢? 在 6.6 节中, 我们匆忙地在 `main()` 函数中加入了对 `;` 与 `q` 的处理, 分别表示“打印”和“退出”。现在, 我们要为这种匆忙付出代价了。重新审视这部分代码:

```
double val = 0;
while (cin) {
    cout << "> ";
    Token t = ts.get();
    if (t.kind == 'q') break;
    if (t.kind == ';')
        cout << "=" << val << '\n';
    else
        ts.putback(t);
    val = expression();
}
```

在上面的代码中, 判断输入是分号后就不再检测 `q`, 而是直接继续调用 `expression()` 函数。`expression()` 函数首先调用 `term()`, `term()` 首先调用 `primary()`, `primary()` 首先检测 `q`。由于字符 `q` 不是 `primary`, 从而输出错误信息。因此, 我们应该在检测完分号以后再对字符 `q` 进行检测。对当前的程序, 我们觉得有必要适当简化程序逻辑, 完整的 `main()` 函数如下:

```
int main()
try
{
    while (cin) {
        cout << "> ";
        Token t = ts.get();
        while (t.kind == ';') t = ts.get(); // eat ';'
        if (t.kind == 'q') {
            keep_window_open();
            return 0;
        }
        ts.putback(t);
        cout << "=" << expression() << endl;
    }
    keep_window_open();
    return 0;
}
catch (exception& e) {
    cerr << e.what() << endl;
    keep_window_open("~~~");
    return 1;
}
catch (...) {
    cerr << "exception\n";
    keep_window_open("~~~");
    return 2;
}
```

改动之后的代码实现了强有力的错误处理机制, 接下来我们就可以开始考虑从其他方面改进计算器程序了。

7.4 处理负数

当你测试完计算器程序后, 你会发现它不能很好地处理负数。例如, 输入 $-1/2$ 将返回一个错误信息, 必须将其写为 $(0-1)/2$, 但这并不符合人们的使用习惯。

在程序调试和测试后期发现这样的问题是很平常的事，只有这时我们才能有机会弄清楚程序到底实现了什么功能，并根据程序给出的反馈不断改进我们的设计。在程序的设计过程中，一种明智的做法是：在安排工作日程时就预留出时间，使我们能有机会体会开发过程中获得的经验教训，从中受益并回头来改进程序。“1.0 版”往往未经必要的精化就发布了，这通常是由于开发日程过紧，或者是为了防止在项目“后期”对详细设计进行修改而采取的呆板的项目管理策略——“后期”添加“特性”的做法将是灾难性的。但实际上，当一个程序对于简单使用来说已经足够好，但还未到可以发布的程度时，开发进程还远未到“后期”。此时还是开发进程的“早期”，正是我们从程序中获取实实在在的经验教训，进行改进的好时机。在实际安排工作日程时，应该将这样的过程考虑其中。

对于本例，我们只需通过修改文法来处理一元减。最简单的方式是修改 `primary` 的定义，将

```
Primary:
    Number
    "(" Expression ")"
```

改为：

```
Primary:
    Number
    "(" Expression ")"
    "-" Primary
    "+" Primary
```

我们还增加了一元加，C++ 语言也支持这个运算符。当有了一元减后，人们通常也会尝试一元加，因此实现它是有意义的。而且既然实现了一元减，实现一元加也很容易，没有必要纠缠于它到底有没有用。于是，实现 `Primary` 的函数变为

```
double primary()
{
    Token t = ts.get();
    switch (t.kind) {
        case '(': // handle '(' expression ')'
        {
            double d = expression();
            t = ts.get();
            if (t.kind != ')') error("'') expected");
            return d;
        }
        case '8': // we use '8' to represent a number
            return t.value; // return the number's value
        case '-':
            return - primary();
        case '+':
            return primary();
        default:
            error("primary expected");
    }
}
```

修改后的 `Primary` 看起来非常简洁，它第一次可以真正地正常运行了。

7.5 模运算：%

当我们最初分析理想中的计算器程序应该具有什么功能时，我们希望它能够处理模运算（也称为取余运算）：%。但 C++ 语言中的模运算符不支持浮点数，因而未加以实现。现在我们可以重新考虑模运算了，可按如下方式简单实现：

1) 添加单词 %。

2) 将 double 型数转换为 int 型, 并使用 % 处理转换后的 int 型数。

为此, 在 term() 函数中增加以下代码:

```
case '%':
{
    double d = term();
    int i1 = int(left);
    int i2 = int(d);
    return i1%i2;
}
```

其中, int(d) 表示将 double 型数据通过截尾方式显式强制转换为 int 型数据, 即简单地将小数点后面的尾数丢弃。不幸的是, 这种转换是多余的 (参见 3.9.2 节), 但我们仍然倾向于进行显式数据类型的转换, 而不是“无意中”将 double 隐式转换为 int。这样, 一切都明确地在我们掌握之中。现在, 计算器程序能正确处理整型数的模运算了。例如:

```
> 2%3;
= 2
> 3%2;
= 1
> 5%3;
= 2
```

如何处理非整型数的模运算? 下面表达式的结果是多少?

```
> 6.7%3.3;
```

目前还没有很好的答案, 因此应禁止对浮点数进行模运算。当检测到参与模运算的浮点数有小数部分时, 就给出错误提示信息。修改后的 term() 函数如下:

```
double term()
{
    double left = primary();
    Token t = ts.get();           // get the next token from Token_stream

    while(true) {
        switch (t.kind) {
            case '*':
                left *= primary();
                t = ts.get();
                break;
            case '/':
                {
                    double d = primary();
                    if (d == 0) error("divide by zero");
                    left /= d;
                    t = ts.get();
                    break;
                }
            case '%':
                {
                    double d = primary();
                    int i1 = int(left);
                    if (i1 != left) error("left-hand operand of % not int");
                    int i2 = int(d);
                    if (i2 != d) error("right-hand operand of % not int");
                    if (i2 == 0) error("%: divide by zero");
                    left = i1%i2;
                    t = ts.get();
                    break;
                }
            default:
                ts.putback(t);      // put t back into the Token_stream
                return left;
        }
    }
}
```

```
    }
  }
}
```

将浮点型数转换为整型数后,可以使用!=来检测数值是否发生变化。如果没有,则表明浮点数没有小数部分,可使用%进行模运算。

我们这里将模运算的操作数限定为整数,是缩小转换(参见3.9.2节和5.6.4节)的变形之一,因此可以使用narrow_cast()函数解决:

```
case '%':
{
    int i1 = narrow_cast<int>(left);
    int i2 = narrow_cast<int>(term());
    if (i2 == 0) error("%: divide by zero");
    left = i1%i2;
    t = ts.get();
    break;
}
```

这个版本非常简短,而且可以说很清晰,但它没有给出恰当的错误提示信息。

7.6 清理代码

我们已经对程序做过几次修改,虽然性能每次都有所提高,但代码却变得有点乱。现在是一个很好的时机来重新检查代码,做适当的清理和简化,并增加一些注释以提高系统的可读性。换句话说,只有当代码达到易于他人接管和维护的状态,程序才算是编写完成。到目前为止,除了缺少注释外,计算器程序总体来说还是不错的,接下来我们进行一点清理工作。

7.6.1 符号常量

回忆一下,我们使用'8'表示单词中包含一个数值。实际上,采用什么值表示数值类型的单词并不重要,只要该值能够与标识其他单词类型的数值区分开即可。不过,这种处理方式使得代码看起来有点古怪,我们应该使用注释语句进行相应的说明。

```
case '8':           // we use '8' to represent a number
    return t.value; // return the number's value
case '-':
    return - primary();
```

老实说,我们也犯过一些错误,比如错敲了'0'而不是'8',因为我们忘记了我们到底选的是哪个值来标识数值型单词。换句话说,直接在代码中用'8'来标识数值型单词是很草率的,而且不容易记住,很容易造成人为错误——实际上'8'就是我们在4.3.1节中曾经提到的应该避免的“魔数”。我们应该引入一个符号常量,来代表这个数:

```
const char number = '8'; // t.kind==number means that t is a number Token
```

const修饰符告诉编译器我们定义了一个值为'8'的字符常量:对number='0',编译器将会给出错误信息。定义了字符常量number以后,我们就不必显式地用'8'来表示数值型单词了。primary函数中的相应代码片段修改如下:

```
case number:
    return t.value; // return the number's value
case '-':
    return - primary();
```

这段代码不再需要任何注释了。实际上,代码本身直接而又清晰地表达出的内容,是任何注释都无法表达清楚的。如果频繁地用注释来解释程序的含义,通常表明你的代码应该改进了。

类似地,Token_stream::get()函数中识别数值的代码修改为:

```

case '.':
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
{   cin.putback(ch);    // put digit back into the input stream
    double val;
    cin >> val;         // read a floating-point number
    return Token(number, val);
}

```

理论上可以为所有的单词设置符号名称,但是太过于繁琐。毕竟,用 '(' 和 '+' 表示左括号和加号这两个单词,是任何人都能够理解的很显然的表示方法。检查计算器程序涉及的单词,只有用 ';' 表示“打印”(或“表达式结束”)以及用 'q' 表示“退出”有些不妥。为什么不用 'p' 和 'e' 呢?在一个大程序中,这种模糊而随意的表示方式迟早会引起问题,所以我们引入如下声明:

```

const char quit = 'q';    // t.kind==quit means that t is a quit Token
const char print = ';';   // t.kind==print means that t is a print Token

```

下面修改 main() 函数中的循环代码:

```

while (cin) {
    cout << "> ";
    Token t = ts.get();
    while (t.kind == print) t = ts.get();
    if (t.kind == quit) {
        keep_window_open();
        return 0;
    }
    ts.putback(t);
    cout << "=" << expression() << endl;
}

```

引入符号名称“print”和“quit”后,提高了代码的可读性。另外,我们并不鼓励人们通过阅读 main() 函数来推测输入什么内容表示“print”和“quit”。例如,如果我们决定用“e”(代表“exit”)表示“quit”,应该是很正常的,而且这一改动应该不用改变 main() 函数中的任何代码。

输入/输出提示符“>”和“=”也同样存在问题,代码中存在这么多概念模糊的符号,如何让初级程序员在阅读 main() 函数时猜测其正确含义?为代码添加注释是一个好主意,但如前所述,引入符号常量更加有效:

```

const string prompt = "> ";
const string result = "=";    // used to indicate that what follows is a result

```

我们想改变输入/输出提示符时该怎么办呢?只需修改这些常量即可,主函数中的循环修改如下:

```

while (cin) {
    cout << prompt;
    Token t = ts.get();
    while (t.kind == print) t = ts.get();
    if (t.kind == quit) {
        keep_window_open();
        return 0;
    }
    ts.putback(t);
    cout << result << expression() << endl;
}

```

7.6.2 使用函数

程序中所使用的函数应该反映出该程序的基本结构,而函数名则应有效地标识代码的逻辑功能模块。到目前为止,计算器程序在这些方面做得还是比较好的: expression()、term() 和 primary() 直接反映出我们对表达式文法的理解,而函数 get() 则用来处理表达式输入和单词识

别。分析一下主函数 `main()`，我们注意到它主要做了两项逻辑上相互独立的任务：

- 1) `main()` 函数搭起了程序的整体框架：启动程序、结束程序、处理致命错误。
- 2) `main()` 函数用一个循环来计算表达式。

理想情况下，一个函数只实现一个独立的逻辑功能（参见 4.5.1 节）。而 `main()` 实现了两个功能，这使得程序结构变得有些模糊。一种显然的改进方法是将表达式计算循环从主函数中分离出来，实现为 `calculate()` 函数：

```
void calculate()           // expression evaluation loop
{
    while (cin) {
        cout << prompt;
        Token t = ts.get();
        while (t.kind == print) t = ts.get();    // first discard all "prints"
        if (t.kind == quit) return;             // quit
        ts.putback(t);
        cout << result << expression() << endl;
    }
}

int main()
try {
    calculate();
    keep_window_open();    // cope with Windows console mode
    return 0;
}
catch (runtime_error& e) {
    cerr << e.what() << endl;
    keep_window_open("~~~");
    return 1;
}
catch (...) {
    cerr << "exception \n";
    keep_window_open("~~~");
    return 2;
}
```

修改后的代码更直接地反映了程序的结构，更易于理解。

7.6.3 代码格式

重新检查一下计算器程序，看看其中是否有“丑陋”的代码，我们发现：

```
switch (ch) {
case 'q': case ';': case '%': case '(': case ')': case '+': case '-': case '*': case '/':
    return Token(ch);    // let each character represent itself
```

在加入对 `'q'`、`'.'` 和 `'%'` 的处理之前，这段代码还不算太坏，但现在变得有些混乱。代码的可读性越差，其中的错误就越难以发现。而这段代码中确实隐藏着一个潜在的 bug！我们修改一下代码，令每行代码只对应 `switch` 语句的一种情况，并加入适当的注释来帮助代码理解，修改后的 `Token_stream` 的 `get()` 函数如下所示：

```
Token Token_stream::get()
// read characters from cin and compose a Token
{
    if (full) {    // check if we already have a Token ready
        full=false;
        return buffer;
    }
```

```

char ch;
cin >> ch;    // note that >> skips whitespace (space, newline, tab, etc.)

switch (ch) {
case quit:
case print:
case '!':
case ')':
case '+':
case '-':
case '*':
case '/':
case '%':
    return Token(ch);    // let each character represent itself
case '.':
    // a floating-point-literal can start with a dot
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':    // numeric literal
{
    cin.putback(ch);    // put digit back into the input stream
    double val;
    cin >> val;    // read a floating-point number
    return Token(number,val);
}
default:
    error("Bad token");
}
}

```

我们当然可以把对每个数字的处理也放在不同的行，但是那样似乎并不能使代码更加清晰，而且导致不能在一屏上显示 `get()` 函数的所有代码。我们理想中的情况是，每个函数的代码都能全部显示在屏幕的可视区域上——在屏幕之外我们无法看到的代码是最有可能隐藏 bug 的地方。因此，代码布局是非常重要的。

另外一个值得注意的地方是，我们在程序中用符号常量 `quit` 替代了字符 `'q'`。这不但提高了程序的可读性，而且保证我们的编程错误会被编译器捕获——如果我们为 `quit` 操作选择的字符与其他单词冲突，将会产生一个编译时错误。

在代码清理阶段，我们有可能意外地引入一些错误。因此，在代码清理之后一定要测试代码的正确性。最好是每做一点改动就测试一次，以便发现错误时，你能记起来是做了什么样的改动导致的这个错误。记住，及早测试、经常测试。

7.6.4 注释

我们在编写计算器程序的过程中加入了一些的注释。好的注释是程序代码的重要组成部分。在程序开发进度很紧时，我们往往会忽略注释。当我们回过头来进行代码清理的时候，是一个很好的时机来全面检查程序的每个部分，检查原来所写的注释是否满足以下要求：

- 1) 在改动了程序代码以后，原来的注释是否仍然有效？
- 2) 对读者来说注释是否充分？（通常是不够充分的。）
- 3) 是否简短清晰，不至于分散读者看代码的注意力？

强调一下最后一条：最好的注释就是让程序本身来表达。如果读者了解程序设计语言，对一些意义已经很明确的代码，就应该避免不必要的冗长注释。例如：

```
x = b+c;    // add b and c and assign the result to x
```

你可能会在本书中发现一些类似的注释，但只限于用来解释你所不熟悉的语言特性的用法。

注释一般用于代码本身很难表达思想的情况。换句话说，代码说明了它做了什么，但没有表

达出它做这些的目的是什么(参见 5.9.1 节)。回顾一下计算器程序,其中就缺少一些必要的注释:函数本身说明了我们是如何处理表达式和单词的,但没有给出表达式和单词的具体含义。对于计算器程序,表达式的文法最适合放入代码注释或者说明文档中,来解释表达式和单词的含义。

```

/*
    Simple calculator

    Revision history:

        Revised by Bjarne Stroustrup May 2007
        Revised by Bjarne Stroustrup August 2006
        Revised by Bjarne Stroustrup August 2004
        Originally written by Bjarne Stroustrup
        (bs@cs.tamu.edu) Spring 2004.

    This program implements a basic expression calculator.
    Input from cin; output to cout.

    The grammar for input is:

    Statement:
        Expression
        Print
        Quit

    Print:
        ;

    Quit:
        q

    Expression:
        Term
        Expression + Term
        Expression - Term

    Term:
        Primary
        Term * Primary
        Term / Primary
        Term % Primary

    Primary:
        Number
        ( Expression )
        - Primary
        + Primary

    Number:
        floating-point-literal

    Input comes from cin through the Token_stream called ts.
*/

```

我们这里使用了块注释,它以`/*`开头,一直到`*/`结束。在注释的开始是程序的版本变化历史,在实际程序中,版本历史一般用于记录每个版本相对于上一个版本做了哪些修正和改进。

注意,注释不是代码。实际上,上面注释中的文法已经进行了简化:对比注释中 Statement 的规则和实际的程序实现可以看出来(参见 7.7 节中的代码)。注释中的规则无法说明 `calculate()` 中的循环语句可以在一次程序执行中计算多个表达式的情况。我们在 7.8.1 节中将对这个问题进行

进一步探讨。

7.7 错误恢复

为什么程序遇到错误就结束运行呢？当初我们选择策略时，这种方式确实看起来简单明了。但是，我们为什么不让程序给出一个错误提示信息，然后继续运行呢？毕竟，我们常常会给出一些小的输入错误，而这并不意味着我们打算结束程序的运行。因此，我们下面尝试为程序加入错误恢复能力。这意味着，程序必须能够捕获异常，并在清理遗留故障后继续运行。

在现在的计算器程序中，所有错误都表示为异常，由 `main()` 函数处理。如果我们希望加入错误恢复功能，必须让 `calculate()` 函数捕获异常，并在计算下一个表达式之前清理故障：

```
void calculate()
{
    while (cin)
    try {
        cout << prompt;
        Token t = ts.get();
        while (t.kind == print) t = ts.get(); // first discard all "prints"
        if (t.kind == quit) return;         // quit
        ts.putback(t);
        cout << result << expression() << endl;
    }
    catch (exception& e) {
        cerr << e.what() << endl;           // write error message
        clean_up_mess();
    }
}
```

我们简单地将 `while` 循环代码块放在 `try-catch` 结构中，`try` 结构在捕获异常后给出错误提示信息，并清理遗留故障。在此之后，程序如往常一样继续运行。

“清理遗留故障”的必要性何在？本质上来说，在错误处理之后准备好继续进行下面的运算，就意味着与错误相关的程序数据都已清理，所有数据都已处于良好的、可预测的状态。在计算器程序中，`Token_stream` 是唯一在函数之外定义的数据。因此，我们所要做的就是清理与错误表达式相关的所有单词，避免它们弄乱下一个表达式。例如：

```
1++2*3; 4+5;
```

这将会引发一个错误，即第二个“+”触发异常之后，`Token_stream` 的缓冲区中仍然保存着 `2 * 3; 4 + 5`，对此有两种处理方式：

- 1) 清除 `Token_stream` 中的所有单词。
- 2) 清除 `Token_stream` 中与当前表达式相关的所有单词。

第一种方式将清除所有单词（包括 `4 + 5;`），而第二种方式只清除 `2 * 3;`，留下 `4 + 5`，随后将会被计算。看起来哪种选择都是合理的，但都会让某些用户感到奇怪。实际上，这两种方式的实现都比较简单，为了简化测试，我们选择第二种处理方式。

在碰到分号之前一直由 `get()` 函数读取输入，并编写如下所示的 `clean_up_mess()` 函数：

```
void clean_up_mess() // naive
{
    while (true) { // skip until we find a print
        Token t = ts.get();
        if (t.kind == print) return;
    }
}
```

不幸的是，这种方式并不能处理所有的情况。考虑如下输入：

```
1@z; 1+3;
```

@ 会导致程序执行 while 循环的 catch 子句，进而调用 `clean_up_mess()` 函数查找下一个分号，然后 `clean_up_mess()` 函数调用 `get()` 函数读入字符 `z`。由于 `z` 不是预定义的单词，这引起了另一个错误，程序转向到 `main()` 函数中的 `catch(...)` 块，最后导致程序结束。太糟糕了！我们还没有机会计算表达式 `1 + 3` 的值，程序就退出了。我们只能再从头开始！

我们可以尝试更精巧的 try 和 catch 结构，但这样会使程序更加混乱。错误处理是程序设计中比较困难的部分，而错误处理过程中发生的错误就更难处理。因此，我们需要设计一种不用抛出异常就能够从 `Token_stream` 中清除字符的方法。在计算器程序中，`get()` 函数是获取输入的唯一途径，但如我们刚刚所见，它遇到错误会抛出一个异常。因此，我们需要设计一个新的操作，很明显，应该将它放在 `Token_stream` 中：

```
class Token_stream {
public:
    Token_stream();           // make a Token_stream that reads from cin
    Token get();              // get a Token
    void putback(Token t);    // put a Token back
    void ignore(char c);      // discard characters up to and including a c

private:
    bool full;               // is there a Token in the buffer?
    Token buffer;            // here is where we keep a Token put back using putback()
};
```

由于 `ignore()` 函数需要检查 `Token_stream` 的缓冲区，因此将其定义为 `Token_stream` 的一个成员函数。我们将“希望得到的内容”作为函数 `ignore()` 的参数，因为毕竟哪些字符可用于错误恢复是上层程序的事情，`Token_stream` 无需了解。我们将这个参数设置为一个字符，是因为希望按原始字符来处理输入，错误恢复过程中提取单词的缺点在前面已经看到了。因此有

```
void Token_stream::ignore(char c)
    // c represents the kind of Token
{
    // first look in buffer:
    if (full && c==buffer.kind) {
        full = false;
        return;
    }
    full = false;

    // now search input:
    char ch = 0;
    while (cin>>ch)
        if (ch==c) return;
}
```

程序首先检查缓冲区，如果缓冲区中的字符就是 `c`，则丢掉它，结束函数；否则一直从 `cin` 中读入字符，直到遇到 `c` 为止。

现在，`clean_up_mess()` 函数可以简化为：

```
void clean_up_mess()
{
    ts.ignore(print);
}
```

错误处理通常是比较困难的。由于很难猜测程序到底会出现什么样的错误，因此需要进行大量的实验与测试。编写一个万无一失的程序对技术要求很高，业余程序员通常会忽略这一方面，因此

高质量的错误处理往往是衡量程序员专业程度的标志。

7.8 变量

我们已经对程序风格和错误处理机制进行了改进,接下来该回过头进一步完善程序的功能了。我们现在已经有一个运行得相当好的程序了,该如何完善它呢?我们希望加入的第一个功能是对变量的支持,变量的加入能够令使用者更好地表示长表达式。类似地,对于科学计算,我们希望支持内置的命名常量,如 π 和 e 等,就像大多数科学计算器那样。

增加变量和常量是对计算器程序的重大扩展,会涉及程序的大部分代码,如果没有充分的理由和时间,最好不要进行这种类型的修改。我们这里为计算器程序增加变量和常量的功能,是因为一方面我们可以借此重新检查程序代码,另一方面还可以学习更多的程序设计技巧。

7.8.1 变量和定义

很明显,对于变量和内置常量来说,最关键的是保存 (name, value) 对,从而通过名字来访问相应的值。可以定义 Variable 类如下:

```
class Variable {
public:
    string name;
    double value;
    Variable (string n, double v) :name(n), value(v) {}
};
```

于是我们就可以使用 name 成员来标识一个 Variable,用 value 成员存储与 name 对应的值。这里给出的构造函数只是为了在使用上符号表示更方便。

我们应该如何存储 Variable 对象,以便能根据 name 来查找 Variable 并存取对应的值?回顾一下我们所学的程序设计工具,最好的方法是使用 Variable 向量:

```
vector<Variable> var_table;
```

我们可以在向量 var_table 中存放任意多个 Variable 对象,当搜索某个给定的 name 时,只要顺序查找向量的每个元素即可。我们可以编写一个函数 get_value(),来实现查找给定的 name,并返回对应的值:

```
double get_value(string s)
// return the value of the Variable named s
{
    for (int i = 0; i < var_table.size(); ++i)
        if (var_table[i].name == s) return var_table[i].value;
    error("get: undefined variable ", s);
}
```

函数代码很简单:遍历 var_table 中的每个 Variable 对象(从向量的第一个元素开始,逐个元素访问,直至最后一个元素),判断变量的 name 成员是否与字符串参数 s 匹配。若匹配,则返回 value 成员中的值。

类似地,我们还可以定义 set_value() 函数实现对 Variable 的赋值:

```
void set_value(string s, double d)
// set the Variable named s to d
{
    for (int i = 0; i < var_table.size(); ++i)
        if (var_table[i].name == s) {
            var_table[i].value = d;
            return;
        }
    error("set: undefined variable ", s);
}
```

现在可以读写 `var_table` 中已存在的变量(描述为 `Variable`)了,但是如何向 `var_table` 增加一个新的 `Variable` 呢? 计算器程序的使用者应该输入什么内容,来定义一个新的变量,以便随后使用它呢? 我们可以考虑采用 C++ 的语法:

```
double var = 7.2;
```

这样是可以的,但计算器程序中的所有变量都是 `double` 类型的,所以这里的“`double`”是可以省略的,变量的定义可简化为:

```
var = 7.2;
```

但是,有时候不能正确区分是新变量声明还是书写错误。例如:

```
var1 = 7.2; // define a new variable called var1  
var1 = 3.2; // define a new variable called var2
```

很明显,我们的原意是 `var2 = 3.2;`,但输入发生了错误(注释没有输入错)。对于这种输入错误导致混淆的情况,我们可以接受它,但更好的方式是遵循程序设计语言(如 C++)的习惯,区分变量的声明(包括初始化)和赋值操作。我们可以使用 `double`,但是在计算器程序中,我们选择更短的关键字 `let` 来定义变量——它实际上来自更老的程序设计语言。

```
let var = 7.2;
```

相关文法如下:

```
Calculation:  
  Statement  
  Print  
  Quit  
  Calculation Statement
```

```
Statement:  
  Declaration  
  Expression
```

```
Declaration:  
  "let" Name "=" Expression
```

`Calculation` 是文法中新增的顶层产生式(规则),表示 `calculate()` 函数中的循环可以在计算器程序的一次运行过程中执行多次计算。它依赖 `Statement` 产生式处理表达式和声明。处理 `Statement` 规则的函数如下:

```
double statement()  
{  
    Token t = ts.get();  
    switch (t.kind) {  
        case let:  
            return declaration();  
        default:  
            ts.putback(t);  
            return expression();  
    }  
}
```

现在可以用 `statement()` 替代 `calculate()` 中的 `expression()`:

```
void calculate()  
{  
    while (cin)  
    try {  
        cout << prompt;  
        Token t = ts.get();  
        while (t.kind == print) t=ts.get(); // first discard all "prints"  
    }  
}
```



```

        if (t.kind == quit) return;           // quit
        ts.putback(t);
        cout << result << statement() << endl;
    }
    catch (exception& e) {
        cerr << e.what() << endl;           // write error message
        clean_up_mess();
    }
}

```

现在我们必须编写 `declaration()` 函数，它应该完成什么功能？应该保证在 `let` 之后出现的是一个 Name 接一个 `=` 再接一个 Expression——也就是语法所描述的形式。应该对 `name` 做何处理？我们应该向向量 `var_table` 中加入一个 `Variable` 对象，其中两个成员分别设置为 `name` 的字符串内容和 `expression` 的值。在随后的表达式计算过程中，我们就可以使用 `get_value()` 和 `set_value()` 函数对变量值进行读写。然而，在动手编写代码之前，我们应该考虑一个问题——如何处理一个变量定义两次的情况？例如：

```

let v1 = 7;
let v1 = 8;

```

一般把这种重复定义作为错误来处理，实际中这往往是拼写错误所致。如本例，我们实际上是想定义两个变量：

```

let v1 = 7;
let v2 = 8;

```

使用变量名 `var` 及值 `val` 定义一个 `Variable`，从逻辑上应该分成两个部分：

- 1) 检查向量 `var_table` 中是否已经存在名为 `var` 的 `Variable`。
- 2) 将 `(var, val)` 加到 `var_table` 中。

这里不支持未初始化的变量。我们定义函数 `is_declared()` 和 `define_name()` 分别实现上述两个独立的逻辑操作：

```

bool is_declared(string var)
    // is var already in var_table?
{
    for (int i = 0; i < var_table.size(); ++i)
        if (var_table[i].name == var) return true;
    return false;
}

double define_name(string var, double val)
    // add (var,val) to var_table
{
    if (is_declared(var)) error(var, " declared twice");
    var_table.push_back(Variable(var,val));
    return val;
}

```

可以通过向量的成员函数 `push_back()` 将一个新的 `Variable` 添加到 `vector < Variable >` 向量中：

```

var_table.push_back(Variable(var,val));

```

其中，`Variable(var, val)` 使用参数 `var` 和 `val` 构造一个 `Variable` 对象，然后 `push_back()` 将它添加到向量 `var_table` 的末尾。假设我们已经能够处理 `let` 和 `name` 单词，可以直接得到函数 `declaration()` 的实现：

```

double declaration()
    // assume we have seen "let"

```

```

// handle: name = expression
// declare a variable called "name" with the initial value "expression"
{
    Token t = ts.get();
    if (t.kind != name) error ("name expected in declaration");
    string var_name = t.name;

    Token t2 = ts.get();
    if (t2.kind != '=') error ("= missing in declaration of ", var_name);

    double d = expression();
    define_name(var_name,d);
    return d;
}

```

注意，我们在函数末尾返回了新变量的值，当初始化表达式比较复杂时，这种方式是比较有用的，例如：

```
let v = d/(t2-t1);
```

这个声明语句定义了变量 `v` 并打印它的值。打印声明变量的值简化了 `calculate()` 函数的代码，因为这样每个 `statement()` 函数就都会返回一个值。一般原则会保持代码简单，而特殊情况会使问题变得复杂。

这种跟踪变量的机制通常称为符号表 (symbol table)，如果使用标准库中的 `map` 的话，这部分代码会得到极大的简化 (参见 21.6.1 节)。

7.8.2 引入单词 name

到现在为止，我们已经对程序进行了很好的改进，但很遗憾，它还不能正常运行。这并不意外，我们对程序下的“第一刀”是不会正常工作的，因为我们甚至还没有完成程序——程序还无法通过编译。程序还不能识别单词 `'='`，但这可以通过在 `Token_stream::get()` 中添加一种情况处理来简单实现。但是对于单词 `let` 和 `name`，必须修改 `get()` 函数来识别这些单词，一种实现如下：

```

const char name = 'a';           // name token
const char let = 'L';             // declaration token
const string declkey = "let";     // declaration keyword

Token Token_stream::get()
{
    if (full) { full=false; return buffer; }
    char ch;
    cin >> ch;
    switch (ch) {
        // as before
    default:
        if (isalpha(ch)) {
            cin.putback(ch);
            string s;
            cin >> s;
            if (s == declkey) return Token(let); // declaration keyword
            return Token(name,s);
        }
        error("Bad token");
    }
}

```

首先请注意函数调用 `isalpha(ch)`，它用来检测输入 `ch` 是否为字符。`isalpha()` 是一个标准库函数，包含在头文件 `std_lib_facilities.h` 中。更多的字符分类函数的内容可以参考 11.6 节。识别变量名

与识别数字的方法是相同的：找到一个正确类别的字符（这里是一个字母）以后，使用 `putback()` 函数把它退回，然后使用 `>>` 读取整个变量名。

不幸的是，程序还是不能通过编译，因为 `Token` 无法存储一个字符串，编译器不能识别 `Token(name, s)`。不过，通过修改 `Token` 的定义就可以解决这个问题：

```
class Token {
public:
    char kind;
    double value;
    string name;
    Token(char ch) : kind(ch), value(0) {}
    Token(char ch, double val) : kind(ch), value(val) {}
    Token(char ch, string n) : kind(ch), name(n) {}
};
```

这里用字符 'L' 表示单词 `let`，字符串 `"let"` 作为关键字。很明显，将关键字改为 `double`、`var`、`#` 或者其他任何字符串都是很容易的，只要将 `declkey` 的值改为想要的字符串，`get()` 函数中读入名字 `s` 后与 `declkey` 比较，就能识别出相应的关键字。

现在重新运行程序，如果输入下列表达式，程序能够正常运行：

```
let x = 3.4;
let y = 2;
x + y * 2;
```

但是，程序不能正确计算下列表达式：

```
let x = 3.4;
let y = 2;
x+y*2;
```

两个例子有什么差别吗？让我们仔细检查一下发生了什么情况。

问题在于我们定义 `Name` 时太马虎了，甚至“忘记”了定义 `Name` 的产生式（参见 7.8.1 节）。根据现有的文法，什么样的字符可以作为名字的一部分呢？字母？当然可以。数字呢？也可以，只要不出现在首字符就可以。那么下划线呢？“+”呢？应该是不允许的，但我们的程序没有正确处理它们。我们还是回过头来认真检查一下代码吧。当读入首字母以后，我们用 `>>` 读入一个字符串。这会把空格以前的所有字符都读取到字符串中。也就是说，`x + y * 2`；虽然是一个表达式，但这里却作为一个变量名处理，甚至分号也成了变量名的一部分。这显然不是我们的本意，也是无法接受的。

我们应该如何修正这个错误呢？首先，我们必须精确地定义 `name` 是什么；然后，我们必须修改 `get()` 函数来实现 `name` 的读取。一个可行的 `name` 的定义如下：以字母开头的字母/数字串，则下面的字符串都是 `name`；

```
a
ab
a1
Z12
asdsdssdfdfdasfda434RTHTD12345dfdsa8fsd888fadsf
```

而下面的字符串都不是：

```
1a
as_s
#
as*
a car
```

当然，按 C++ 的语法，`as_s` 是一个合法的 `name`。可将 `get()` 函数的 `default` 项修改如下：

```

default:
    if (isalpha(ch)) {
        string s;
        s += ch;
        while (cin.get(ch) && (isalpha(ch) || isdigit(ch))) s+=ch;
        cin.putback(ch);
        if (s == declkey) return Token(let); // declaration keyword
        return Token(name,s);
    }
    error("Bad token");

```

新的代码，将原来直接将字符串读入 *s* 的方式，改为不断读入字符，只要是字母或数字，就添加到 *s* 的末尾（语句 *s += ch* 表示将字符 *ch* 添加到字符串 *s* 的末尾）。while 语句看起来很奇怪：

```
while (cin.get(ch) && (isalpha(ch) || isdigit(ch))) s+=ch;
```

这条语句中使用 *cin* 的成员函数 *get()* 读一个字符到 *ch*，并检查是否为字母或数字。如果是，就将 *ch* 添加到 *s* 的末尾，然后继续读入下一个字符。成员函数 *get()* 与 *>>* 的作用相似，只是它默认情况下不会在遇到空格时停止。

7.8.3 预定义名字

现在程序已经支持名字了，我们可以预定义一些常用的名字。例如，假如我们的计算器程序用于科学计算，那么我们可能需要预定义 *pi* 和 *e*。我们应该在程序中什么位置放置这些定义？可以放在 *main()* 函数中的 *calculate()* 函数调用以前，或者放在 *calculate()* 函数中的计算循环之前。由于这些定义不是任何表达式计算的组成部分，因此可以将它们放在 *main()* 函数中。

```

int main()
try {
    // predefine names:
    define_name("pi",3.1415926535);
    define_name("e",2.7182818284);

    calculate();

    keep_window_open();    // cope with Windows console mode
    return 0;
}
catch (exception& e) {
    cerr << e.what() << endl;
    keep_window_open("~~~");
    return 1;
}
catch (...) {
    cerr << "exception \n";
    keep_window_open("~~~");
    return 2;
}

```

7.8.4 我们到达目的地了吗

显然没有，在对程序做了诸多修改以后，还需要对程序进行测试、清理代码和修改注释等。而且，还可以定义更多的操作和变量。例如，我们还没有在程序中提供赋值操作符（参见练习 2），如果实现执行赋值操作的话，我们还应该区分变量和常量（参见练习 3）。

我们先回到最初不支持命名变量的计算器程序，仔细回顾一下实现命名变量功能的代码，可能会有两种不同的反应：

- 1) 实现变量并不那么糟，大概用三十几行代码就可以了。
- 2) 实现变量是一个重大的扩展，几乎涉及每个函数，并且在计算器程序中引入了一个全新的

概念。在没有实现赋值操作的情况下，代码量已经增加了大约45%！

计算器程序是我们第一个比较复杂的程序，站在这个角度来看，第二种反应是比较合理的。一般来说，如果一个改进程序的建议会使程序的代码量和复杂度增加50%左右，第二个反应是很正常的。如果真按这样的建议去做了，你会发现整个过程更像是基于原来版本重写了一个新的程序。而且，你最好就把这个过程当做重写程序来对待，这样会有更好的效果。特别地，如果我们能够分阶段编写和测试程序，就像设计计算器程序这样，我们最好就这么做，这比一下子就编写完整的程序要好得多。

简单练习

1. 从程序 `calculator08buggy.cpp` 开始，修改其中的错误，使之能编译通过。
2. 阅读整个计算器程序并添加适当的注释。
3. 你会发现程序中存在一些错误（我们特意加入了一些不明显的错误让你来查找），这些错误都未在本章中出现过，尝试修正它们。
4. 测试：准备一组测试数据，用来测试计算器程序。要注意测试用例的完整性，思考你要通过测试用例查找什么？测试用例应包括负数、0、非常小的数、非常大的数和一些“愚蠢”输入。
5. 进行测试，并修改在阅读代码过程中没有发现的错误。
6. 增加一个预定义名字 `k`，其值为 1000。
7. 为用户提供平方根函数 `sqrt()`，如允许用户计算 `sqrt(2 + 6.7)`。`sqrt(x)` 的值是 x 的平方根，例如 `sqrt(9)` 的值为 3。使用标准库函数 `sqrt()` 完成平方根的计算，这个函数包含在头文件 `std_lib_facilities.h` 中。记得更新程序代码注释以及文法。
8. 捕获求负数的平方根的异常，并给出适当的错误信息。
9. 增加幂函数 `pow(x, i)`，例如 `pow(2.5, 3)` 表示 $2.5 * 2.5 * 2.5$ ，要求 i 为整数，可使用与 `%` 运算符相同的方法处理。
10. 将“声明关键字”`let` 改为 `#`。
11. 将“退出关键字”`q` 改为 `exit`，与“`let`”的处理一样，我们需要定义一个表示“退出”的字符串，参见 7.8.2 节。

思考题

1. 为什么还要对程序的第一版做这些改进？给出几条原因。
2. 为什么输入表达式“`1 + 2; q`”后，程序没有退出而是给出一个错误信息？
3. 为什么选择用一个字符常量表示 `number`？
4. 为什么把 `main()` 函数分成两个相互独立的部分，分别实现了什么功能？
5. 为什么把程序代码分成若干个小函数？试阐明划分原则。
6. 代码注释的目的是什么？如何为程序增加注释？
7. `narrow_cast` 的作用是什么？
8. 符号常量的用途是什么？
9. 为什么关心代码布局？
10. 如何处理浮点数的模运算（`%`）？
11. `is_declared()` 函数的功能是什么？它是如何工作的？
12. `let` 单词对应的输入内容是由多个字符构成的，在修改后的程序中，如何将其作为一个单词读入？
13. 计算器程序中的 `name` 可以是什么形式，不能是什么形式，对应的规则是怎样的？
14. 为什么说以增量方式设计程序是一个比较好的主意？
15. 什么时候开始对程序进行测试？
16. 什么时候对程序进行再测试？
17. 如何决定函数的划分？

- 18. 为什么添加代码注释?
- 19. 注释中应该写些什么内容, 不应该写什么内容?
- 20. 什么时候可以认为已经完成了一个程序?



术语

代码布局	程序维护	程序框架	注释	错误恢复
符号常量	错误处理	版本历史	测试	功能蔓延



习题

- 1. 修改计算器程序, 允许名字中出现下划线。
- 2. 提供赋值操作符“=”, 以便在 let 定义变量后能够修改其值。
- 3. 提供命名常量——不允许更改其值。提示: 在 Variable 中增加一个用于区分常量和变量的成员, 并根据该成员判断 set_value() 是否允许执行。如果允许用户定义常量(而不是计算器程序中预定义的 pi 和 e 那样的常量), 必须增加一个符号, 用户可用其表达常量定义, 例如用“const”表示常量定义: const pi = 3.14;。
- 4. get_value()、set_value()、is_declared() 和 declare_name() 等函数都可以操作全局变量 var_table。定义一个 Symbol_table 类, 其成员为 vector < Variable > 类型的 var_table, 成员函数为 get()、set()、is_declared() 和 declare(), 并使用该类重新编写计算器程序。
- 5. 修改 Token_stream::get() 函数, 在读到换行时返回单词 print。这就需要寻找空格并对换行(‘\n’)进行特殊处理。可以使用标准库函数 isspace(ch), 当 ch 为空格时返回真值。
- 6. 每个程序都应该具有给用户提供帮助信息的功能, 修改计算器程序, 当用户输入“H”(大写或小写均可)时能够显示帮助信息。
- 7. 将命令符“q”和“h”分别改为“quit”和“help”。
- 8. 7.6.4 节给出的文法是不完整的(我提醒过你不要过分依赖注释), 没有定义语句序列, 例如 4 + 4; 5 - 6;, 也不包括 7.8 节对文法所做的改进。修改文法, 并为注释添加你认为必要的内容。
- 9. 定义一个 Table 类, 其成员为 vector < Variable > 类型的变量, 成员函数为 get()、set() 和 declare(), 并在计算器程序中用 Table 类对象 symbol_table 替换 var_table。
- 10. 为计算器程序提出三种本章没有提及的改进, 并实现其中的一种。
- 11. 修改计算器程序, 使其仅仅能处理整数, 并在发生上溢和下溢时给出错误信息。
- 12. 实现赋值操作符, 以便能修改初始化以后的变量的值。试讨论赋值操作符的用处以及可能引起的问题。
- 13. 回顾你在做第 4 章和第 5 章习题时所编写的两个程序, 根据本章给出的原则清理代码, 看看在这个过程中能否发现错误。



附言

很偶然地, 我们通过一个简单的例子了解了编译器是如何工作的。计算器程序分析输入流, 将其分解为单词, 并根据文法规则理解其意义。这正是编译器所做的工作。在分析了前面阶段的输入之后, 编译器将生成另外一种形式的描述(目标代码), 可供随后执行。而计算器程序分析了表达式的含义后, 会立刻计算其值, 这种程序一般称为解释器, 而不是编译器。

第8章 函数相关的技术细节

“再多的天赋也战胜不了对细节的偏执。”

——俗语

在本章和第9章中，我们将注意力从程序设计转移到我们主要的编程工具——C++上。我们会介绍一些语言的技术细节，来给出一个C++的基本功能的稍宽的视角，并从更系统化的角度讨论这些功能。这两章还会回顾很多前面章节介绍过的程序设计概念，并提供一个研究语言工具的机会，这两章不介绍新的程序设计技术和概念。

8.1 技术细节

如果可以选择的话，我们更愿意讨论程序设计，而不是程序设计语言的特性；也就是说，我们认为，如何用代码表达思想远比我们用来表达这些思想的程序设计语言技术细节更有意思。自然语言中也有类似的情况：我们更愿意讨论一部好的小说中的思想和它表达这些思想的方式，而不愿意研究其中的语法和词汇。总之，重要的是思想和如何用代码表达思想，而不是单独的编程语言特性。

但是，我们不是总可以选择。当你开始编程时，你用的编程语言对你而言就是一门外语，你必须学习它的“语法和词汇表”。这就是本章和第9章要做的事情，但请不要忘记：

- 我们要学习的主要是程序设计。
- 我们生产的是程序和系统。
- 程序设计语言(只)是工具。

记住这一点似乎极其困难，很多程序员对明显是编程语言语法和语义次要细节的东西表现出巨大的热情。特别是，很多人有这样一种错误观念：他们使用的第一种编程语言中的工作方法是做任何事的“不二法门”，请不要掉入这种陷阱。C++在很多方面是一种非常好的编程语言，但它并不完美，任何其他编程语言也都一样。

大多数程序设计概念是通用的，而很多这种概念被流行的程序设计语言所广泛支持。这意味着，我们在一门好的程序设计课中学到的基本思想和技术可以从一种编程语言延续到下一种语言。也就是说，它们可以(不同程度地)方便地用于所有语言。而编程语言的技术细节，则只限于特定的语言。幸运的是，编程语言不是在真空中设计出来的，因此你在这里学到的很多内容都会在其他语言中找到明显的对应内容。特别是C++与C(参见第27章)、Java和C#属于同一类语言，它们具有相当多的共性。

注意，当讨论语言技术问题时，我们故意使用非描述性的命名，如f、g、X和y。我们这样做是为了强调这些例子的技术性质，保证这些例子很简短，以及尽力避免将语言技术细节和真正的程序设计方法混淆而令你困惑。当你看到非描述性的名字(例如那些永远不应该用在真实代码中的名字)时，请将注意力放在代码的语言技术层面上来。典型的语言技术实例由仅仅用来展示语言规则的代码组成。如果你编译并运行这些代码，你会得到很多“变量未使用”的警告，而这样的技术性程序片段很少具有实用意义。

请注意我们在本章介绍的内容不是 C++ 语法和语义的完整描述，甚至不是我们介绍过的 C++ 功能的完整描述。ISO C++ 标准的篇幅有 756 页，充满晦涩的技术语言；而 Stroustrup 所著的《The C++ Programming Language》一书有 1000 多页，针对的是有经验的程序员。本书不会在完备性和综合性方面与它们一争高下，本书的优势是易懂，值得阅读。

8.2 声明和定义

声明 (declaration) 语句将名字引入作用域 (参见 8.4 节)，其作用是：

- 为命名实体 (如变量、函数) 指定一个类型。
- (可选) 进行初始化 (例如，为变量指定一个初始值或为函数指定函数体)。

下面即为声明语句的例子：

```
int a = 7;           // an int variable
const double cd = 8.7; // a double-precision floating-point constant
double sqrt(double); // a function taking a double argument
                    // and returning a double result
vector<Token> v;     // a vector-of-Tokens variable
```

C++ 程序中的名字都必须先声明后使用。考虑下面代码：

```
int main()
{
    cout << f(i) << '\n';
}
```

编译器最少会给出三个“未声明的标识符”的错误，因为在此程序片段中任何地方都没有 `cout`、`f` 和 `i` 的声明。我们可以通过包含头文件 `std-lib-facilities.h` 得到 `cout` 的声明：

```
#include "std_lib_facilities.h" // we find the declaration of cout in here

int main()
{
    cout << f(i) << '\n';
}
```

现在，只剩两个“未定义”错误了。当编写实用程序时，你会发现大多数声明都是在头文件中给出的。一般来说，对于在“其他地方”定义的有用的功能，可以在头文件中为其定义接口。大致来说，一个声明定义了一些功能的使用方式，也就是定义了函数、变量或类的接口。请注意声明的这种用途有一个明显的但易被忽视的优点：我们不必了解 `cout` 及其 `<<` 操作符的定义的细节，我们只需用 `#include` 包含它们的声明即可。我们甚至无需真正了解它们的声明，从教材、手册、代码实例或其他资料中了解 `cout` 应如何使用就够了。编译器会读取头文件中的声明，以便“理解”我们的程序。

现在，我们还需要声明 `f` 和 `i`，可以这样做：

```
#include "std_lib_facilities.h" // we find the declaration of cout in here

int f(int); // declaration of f

int main()
{
    int i = 7; // declaration of i
    cout << f(i) << '\n';
}
```

这段代码就可以编译通过了，因为每个名字都已经声明过了，但还会连接失败 (参见 2.4 节)，因为我们还没有定义函数 `f()`；也就是说，我们还没有指出 `f()` 实际上应该做什么。

如果一个声明(还)给出了声明的实体的完整描述,我们则称之为定义(definition)。

下面是一个定义的例子:

```
int a = 7;
vector<double> v;
double sqrt(double d) { /* ... */ }
```

每个定义同时也是一个声明,但某些声明不是定义。下面列出一些不是定义的声明,每个声明都需要在代码的其他位置给出对应的定义。

```
double sqrt(double);    // no function body here
extern int a;           // "extern plus no initializer" means "not definition"
```

当我们对比定义和声明时,我们按惯例用“声明”表示“不是定义的定义”,即使这有点不那么严谨。

一个定义确切指明了一个名字代表什么。特别地,一个变量定义会为该变量分配内存空间。

因此,你不能重复定义某个名字,例如:

```
double sqrt(double d) { /* ... */ } // definition
double sqrt(double d) { /* ... */ } // error: double definition
```

```
int a; // definition
int a; // error: double definition
```

相反,一个非定义声明仅仅告诉你如何使用一个名字,它只是一个接口,不会为变量分配内存空间或为函数指定函数体。因此,你可以声明一个名字任意多次,只要一致即可:

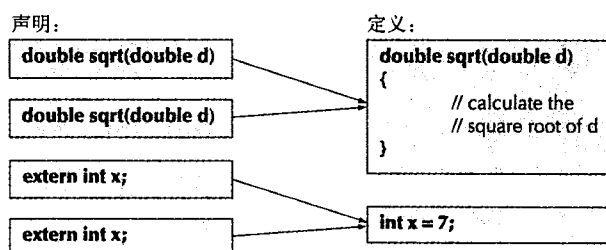
```
int x = 7;           // definition
extern int x;        // declaration
extern int x;        // another declaration

double sqrt(double); // declaration
double sqrt(double d) { /* ... */ } // definition
double sqrt(double); // another declaration of sqrt
double sqrt(double); // yet another declaration of sqrt
```

```
int sqrt(double);    // error: inconsistent declarations of sqrt
```

最后一个声明为什么是错误的? 因为不允许有两个函数都叫 `sqrt`, 接受一个同样是 `double` 类型的参数, 但却返回不同类型的值(`int` 和 `double`)

`x` 的第二个声明中使用的关键字 `extern` 表示此声明不是一个定义。这个关键字几乎没什么用处, 我们建议你不要使用它, 但是你会在别人的代码中看见它, 特别是那些使用了非常多全局变量的代码(参见 8.4 节和 8.6.2 节)。



为什么 C++ 既提供声明功能, 又提供定义功能呢? 这两者间的区别反映出“如何使用一个实体(接口)”与“这个实体如何完成它应该做的事情(实现)”之间的根本区别。对于一个变量来说, 其声明仅仅提供了类型, 只有定义才能提供对象(内存空间)。对于一个函数

来说,其声明也是只提供了类型(参数类型和返回类型),只有定义才提供函数体(可执行的语句)。注意,函数体是作为程序的一部分保存在内存中的,因此可以说函数和变量定义消耗了内存,而声明却没有。

声明和定义之间的区别使我们可以将一个程序分为很多部分,分开编译。声明功能使程序的每个部分都能保有程序其他部分的一个视图,而不必关心其他部分中的定义。所有声明(包括唯一的那个定义)必须一致,而整个程序中实体的命名也应该一致。我们将在 8.3 节中进一步讨论这个问题。在此,我们只是提醒你回顾一下第 6 章中的表达式分析器:其中 `expression()` 调用 `term()`, `term()` 调用 `primary()`, 而 `primary()` 又调用了 `expression()`。由于在 C++ 程序中每个名字都要先声明再使用,所以简单地定义这三个函数是行不通的:

```
double expression();    // just a declaration, not a definition
```

```
double primary()
{
    //...
    expression();
    //...
}
```

```
double term()
{
    //...
    primary();
    //...
}
```

```
double expression()
{
    //...
    term();
    //...
}
```

我们可以按任意顺序排列这四个函数,无论如何必然会有一个函数调用在其后定义的函数。这里,我们需要“前置声明”(forward declaration)。因此,在 `primary()` 的定义前声明 `expression()`, 这样就一切顺利了。在实际编程中,这种循环调用非常常见。

为什么名字需要在使用前声明呢?为什么我们不能要求编译器通过读程序(就像我们做的那样)找出定义,来获得函数应该如何调用的信息呢?我们当然可以这样要求,但这会导致“有趣”的技术问题,所以我们决定不这样做。C++ 规范要求声明在使用前定义(类成员除外,参见 9.4.4 节)。毕竟,这种方式已经是一般写作(不是编写程序)的惯例了:当你阅读一本教材时,你当然希望作者在使用一个术语之前先定义它;否则,你不得不一直去猜测术语的含义或去查索引。无论是对人还是编译器,“先声明后使用”的原则简化了阅读。在一个程序中,“先使用后定义”之所以重要还有另外一个原因。在一个几千行(甚至几十万行)的程序中,很多我们希望调用的函数都是在“别处”定义过的。而这个“别处”,我们往往不希望知道到底是哪。只需了解我们使用的实体的声明,把编译器从阅读大量程序文本中解脱出来。

8.2.1 声明的类别

C++ 允许程序员定义很多类别的实体,我们比较关心的有:

- 变量
- 常量

- 函数(参见 8.5 节)
- 命名空间(参见 8.7 节)
- 类型(类和枚举, 参见第 9 章)
- 模板(参见第 19 章)

8.2.2 变量和常量声明

一个变量或常量声明需指定名字和类型, 并可进行初始化。例如:

```
int a;           // no initializer
double d = 7;    // initializer using the = syntax
vector<int> vi(10); // initializer using the () syntax
```

你可在 Stroustrup 的《The C++ Programming Language》一书中和 ISO C++ 标准中找到变量和常量声明的完整语法。

常量声明的语法与变量声明一样, 差别在于类型之前多了一个关键字 `const`, 而且必须进行初始化:

```
const int x = 7;    // initializer using the = syntax
const int x2(9);    // initializer using the () syntax
const int y;        // error: no initializer
```

必须进行初始化的原因是显然的: 如果一个常量没有值的话, 它何以为常量呢? 对变量也进行初始化通常是个好主意, 未初始化的变量常会导致隐蔽的错误。例如:

```
void f(int z)
{
    int x; // uninitialized
    // ... no assignment to x here ...
    x = 7; // give x a value
    // ...
}
```

这段代码看起来再正常不过了, 但如果在第一个“...”处包含对 `x` 的使用又如何呢? 例如:

```
void f(int z)
{
    int x; // uninitialized
    // ... no assignment to x here ...
    if (z > x) {
        // ...
    }
    // ...
    x = 7; // give x a value
    // ...
}
```

因为 `x` 未初始化, 所以执行 `z > x` 的结果是未定义的。在不同的机器平台上, 比较操作 `z > x` 会给出不同的结果, 甚至同一台机器上执行多次也会给出不同的结果。原则上, `z > x` 应导致程序因一个硬件错误而终止, 但多数时候这不会发生, 取而代之的是我们会得到一个不可预知的结果。

我们自然不会故意这么做, 但我们可能犯错误, 如果没有坚持初始化变量, 上述情况就会发生。记住, 很多“愚蠢的错误”(比如对于一个未初始化的变量, 在对其赋值之前就使用它)都是在你很忙之后疲倦的时候发生的。编译器会尽力给出警告, 但对于复杂的代码(这类错误最可能发生的地方)编译器还无力捕捉所有这种错误。有的人不习惯初始化变量, 这通常是因为他们学习程序设计所用的语言不允许或不鼓励一致的初始化; 因此你会在别人的代码中看到这样的例子。请不要因为忘记初始化你自己定义的变量, 而向你的程序中引

入错误。

8.2.3 默认初始化

你可能已经注意到了，我们通常不对字符串、向量等对象进行初始化。例如：

```
vector<string> v;
string s;
while (cin>>s) v.push_back(s);
```

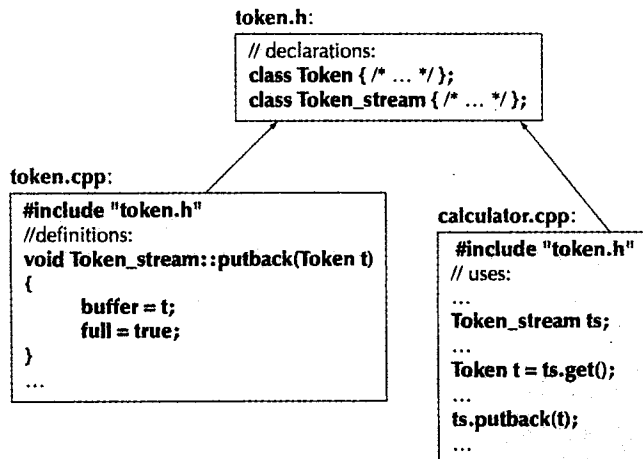
这并不是“变量必须先初始化再使用”这条规则的例外情况。之所以出现这种情况，是因为我们定义 `string` 类和 `vector` 类时定义了默认初始化机制，如果代码中不显式进行初始化，这两种对象就会被用一个默认值进行初始化。因此，上述代码执行到循环时，`v` 的值为空（不包含任何元素），`s` 的值为空串（“”）。保证默认初始化的机制称为默认构造函数，参见 9.7.3 节。

不幸的是，C++ 不允许我们对内置类型设置默认初始化功能。全局变量会被默认初始化为 0，但你应该尽量少用全局变量。而最常使用的变量、局部变量和类成员，是不会被初始化的，除非你提供相应的初始化程序（或默认构造函数）。我们已经提示过了，你在实践中一定要注意！

8.3 头文件

我们如何管理声明和定义呢？这是一个大问题，因为声明和定义要一致，而实际程序可能会有数万个声明，甚至几十万个也不罕见。一般地，当编写程序时，我们使用的定义多数都不是我们写的。例如，`cout` 和 `sqrt()` 的实现就是别人在很多年前写的，我们只是使用它们。

在 C++ 中，对于“别处”定义的功能的声明，管理它们的关键是“头”。本质上，一个头（header）是一个声明的集合，一般定义于一个文件，因此也称为头文件（header file）。这样的头文件用 `#include` 包含在我们的源文件中。例如，我们可能决定改进的计算器程序（参见第 6 章和第 7 章）的源码组织，将单词管理部分分隔出去。我们可以定义一个包含 `Token` 类和 `Token_stream` 类的声明的头文件 `token.h`：



`Token` 和 `Token_stream` 的声明在头文件 `token.h` 中，而它们的定义在 `token.cpp` 中。后缀 `.h` 通常用于 C++ 头文件，而 `.cpp` 后缀通常用于 C++ 源文件。实际上，C++ 语言并不关心文件后缀，但一些编译器和很多程序开发环境坚持这种命名习惯，因此你的源码组织也请遵循这

一惯例。

原则上, `#include "file.h"` 只是简单地将 `file.h` 中的声明复制到你的文件中 `#include` 指令处。

例如, 我们可以写一个头文件 `f.h`:

```
// f.h
int f(int);
```

并将它包含于我们的源文件 `user.cpp` 中:

```
// user.cpp
#include "f.h"
int g(int i)
{
    return f(i);
}
```

当编译 `user.cpp` 时, 编译器会执行包含操作, 然后编译得到的如下程序:

```
int f(int);
int g(int i)
{
    return f(i);
}
```

由于 `#include` 的处理逻辑上在编译器的任何其他动作之前进行, 因此称为预处理 (preprocessing) (参见 A.17 节)。

为了方便一致性检查, 我们在使用声明的源文件和给出定义的源文件中都包含头文件。这样, 编译器就能尽可能快地捕获错误。例如, 假定实现 `Token_stream::putback()` 的程序员犯了如下错误:

```
Token Token_stream::putback(Token t)
{
    buffer.push_back(t);
    return t;
}
```

这段代码看起来没有问题(虽然它存在错误), 幸运的是, 编译器可以发现这个错误, 因为它看到了(包含进来的) `Token_stream::putback()` 的声明。将之与这里的定义比较后, 编译器发现 `putback()` 不应该返回一个 `Token`, 另外 `buffer` 是一个 `Token` 而不是一个 `vector<Token>`, 因此我们不能在其上使用 `push_back()`。这个错误产生的原因是, 我们在原有的代码上继续工作, 试图改进它, 但进行的修改没有保证整个程序的一致性。

类似地, 考虑下面代码中的错误:

```
Token t = ts.gettt(); // error: no member gett
// ...
ts.putback(); // error: argument missing
```

编译器会立即报告错误, 因为头文件 `token.h` 给出了一致性检查所需的所有信息。

头文件 `std-lib-facilities.h` 包含了我们所使用的标准库中的功能的声明, 如 `cout`、`vector` 和 `sqrt()`, 以及一些不在标准库中的简单工具函数的声明, 如 `error()`。在 12.8 节中我们会说明如何直接使用标准库头文件。

一个头文件通常会被包含在很多个源文件中, 这意味着头文件只能包含那些可以在多个文件中重复多次的声明(如函数声明、类定义和数值常量的定义)。

8.4 作用域

作用域 (scope) 是一个程序文本区域。每个名字都定义在一个作用域中，在声明点到作用域结束的区间内有效。例如：

```
void f()
{
    g();    // error: g() isn't (yet) in scope
}

void g()
{
    f();    // OK: f() is in scope
}

void h()
{
    int x = y;    // error: y isn't (yet) in scope
    int y = x;    // OK: x is in scope
    g();          // OK: g() is in scope
}
```

名字在其声明的定义域嵌套的定义域中也有效。例如，上面代码中对 `f()` 的调用在 `g()` 的作用域中，此作用域嵌套于全局作用域，全局作用域不在任何其他作用域内。名字必须先声明后使用的规则这里还是适用的，因此 `f()` 不能调用 `g()`。

C++ 支持多种类型的作用域，帮助我们控制变量在哪里可用：

- 全局作用域：在任何其他作用域之外的程序区域。
- 名字空间作用域：一个名字空间作用域嵌套于全局作用域或另一个名字空间作用域中，参见 8.7 节。
- 类作用域：一个类内的程序区域，参见 9.2 节。
- 局部作用域：位于 `{ ... }` 大括号之间或函数参数列表中的程序区域。
- 语句作用域：例如，`for` 语句内的程序区域。

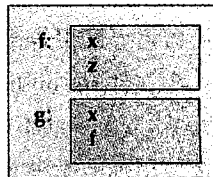
作用域的主要作用是保持名字的局部性，使之不影响声明于其他地方的名字。例如：

```
void f(int x)    // f is global; x is local to f
{
    int z = x+7; // z is local
}

int g(int x)    // g is global; x is local to g
{
    int f = x+2; // f is local
    return 2*f;
}
```

下图描述了上面代码中的作用域信息。这里，`f()` 中的 `x` 与 `g()` 中的 `x` 是不一样的。它们不会冲突，因为不在同一个作用域中：`f()` 中的 `x` 在 `f` 的局部作用域中，而 `g()` 中的 `x` 在 `g` 的局部作用域中。位于同一个作用域中，不能共存的两个声明称为冲突 (clash)。类似地，`g()` 中定义的 `f` (显然) 不是全局函数 `f()`。

全局作用域：



下面代码中局部作用域的使用与上例类似，但这段代码更接近实际的例子：


```

int max(int a, int b)           // max is global; a and b are local
{
    return (a>=b) ? a : b;
}

int abs(int a)                  // not max()'s a
{
    return (a<0) ? -a : a;
}

```

在标准库中你会发现 `max()` 和 `abs()`，因此你不必自己编写这两个函数。?: 结构称做算术 if (arithmetic if) 或条件表达式 (conditional expression)。当 $a \geq b$ 时， $(a \geq b) ? a : b$ 的值为 a ，否则为 b 。条件表达式可以帮助我们避免下面这种冗长的代码：

```

int max(int a, int b)           // max is global; a and b are local
{
    int m;                       // m is local
    if (a>=b)
        m = a;
    else
        m = b;
    return m;
}

```

因此，除了很明显的全局作用域外，其他作用域都使名字保持局部性。在很多场合，局部性是一种很好的性质，因此你应该尽量保持名字的局部性。当你在函数、类和名字空间内部声明变量和函数时，它们不会影响你的变量和函数。记住，实际程序中有成千上万的命名实体，为了使这种程序易于管理，多数名字都应该是局部性的。

下面是一个较大的实例，说明了名字是如何在语句和语句块 (包括函数体) 末尾离开作用域的：

```

// no r, i, or v here
class My_vector {
    vector<int> v;                // v is in class scope
public:
    int largest()
    {
        int r = 0;               // r is local (smallest nonnegative int)
        for (int i = 0; i<v.size(); ++i)
            r = max(r,abs(v[i])); // i is in the for's statement scope
        // no i here
        return r;
    }
    // no r here
};
// no v here

int x;    // global variable — avoid those where you can
int y;

int f()
{
    int x;    // local variable
    x = 7;    // the local x
    {
        int x = y;    // local x initialized by global y
        ++x;          // the x from the previous line
    }
    ++x;    // the x from the first line of f()
    return x;
}

```

只要可能,你应该避免这种复杂的嵌套和隐藏。记住:“保持简单性!”

一个名字的作用域越大,名字就应该越长、越有描述性:将全局变量命名为 `x`、`y` 和 `f` 是灾难性的。你在程序中应尽量少用全局变量,一个主要原因是你很难知道哪个函数会修改它们。在大的程序中,基本不可能知道哪个函数修改了一个全局变量。想象你正在调试一个程序,而你发现一个全局变量的值与预期不符。那么是谁赋予它这个值?为什么这样赋值?是哪个函数干的?你如何能知道这些信息?赋予此变量这个错误值的函数可能在你从未见过的一个源文件中!如果非有不可的话,一个好的程序也应该只有非常少(如一个或两个)的全局变量。例如,我们在第6章和第7章给出的计算器程序只有两个全局变量:单词流 `ts` 和符号表 `names`。

注意,多数 C++ 语法结构定义了嵌入的作用域:

- 类中的函数:成员函数(参见 9.4.2 节)

```
class C {
public:
    void f();
    void g()    // a member function can be defined within its class
    {
        // ...
    }
    // ...
};

void C::f()    // a member definition can be outside its class
{
    // ...
}
```

这是一种最常见和最有用的情况。

- 类中的类:成员类(也称做嵌入类)

```
class C {
public:
    struct M {
        // ...
    };
    // ...
};
```

这只在复杂类中才有用,记住理想情况是保持类简短、简单。

- 函数中的类:局部类

```
void f()
{
    class L {
        // ...
    };
    // ...
}
```

应避免这种代码,如果你觉得需要一个局部类,那么你的函数可能太长了。

- 函数中的函数:局部函数(也称做嵌套函数)

```
void f()
{
    void g()    // illegal
    {
        // ...
    }
    // ...
}
```

这在 C++ 中是不合法的，不要写这种代码，编辑器会拒绝它。

- 函数或块中的块：嵌套块

```
void f(int x, int y)
{
    if (x>y) {
        // ...
    }
    else {
        // ...
        {
            // ...
        }
        // ...
    }
}
```

嵌套块是避免不了的，但要对复杂的嵌套保持警惕：它很容易隐藏错误。

C++ 还提供了一种语言特性：名字空间，专门用于表达作用域，参见 8.7 节。

注意，我们使用一致的缩进格式来表明嵌套，如果不这样，嵌套的结构就会很难读。例如：

```
// dangerously ugly code
struct X {
void f(int x) {
struct Y {
int f() { return 1; } int m; };
int m;
m=x; Y m2;
return f(m2.f()); }
int m; void g(int m) {
if (m) f(m+2); else {
g(m+2); }}
X() {} void m3() {
}

void main() {
X a; a.f(2);
};
```

难读的代码往往隐藏着错误。当你使用某种 IDE 时，IDE 会尽力自动地（根据某种“恰当的”规则）将你的代码整理成恰当的缩进格式。也有一种“代码美化器”，可以重整源代码文件的格式（通常可以允许你自己选择格式）。但是，令你的代码可读的最终的责任还是在你自己。

8.5 函数调用和返回

函数为我们提供了表示操作和计算的途径。当我们要完成某些工作，而这些工作值得起一个名字，我们就可以编写一个函数。C++ 语言为我们提供了运算符（如 + 和 *），我们可以在表达式中用运算符从运算对象产生出新值。C++ 还提供了语句（如 for 和 if），我们可以用它们控制程序执行的顺序。为了组织由这些原语构成的代码，我们需要使用函数。

为了完成自身的工作，函数通常需要参数，很多函数还返回一个结果。本节关注参数如何指定和传递。

8.5.1 声明参数和返回类型

函数是 C++ 中我们用来命名和表示计算和操作的语法结构。一个函数声明由一个返回值后跟函数名，再接一个形式参数（formal argument，简称形参）列表构成。例如：

```
double fct(int a, double d);           // declaration of fct (no body)
double fct(int a, double d) { return a*d; } // definition of fct
```

一个函数定义包含函数体(调用此函数时应执行的语句),而一个非定义的函数声明则只接一个分号。形参(parameter)通常也称为参数。如果你不希望一个函数接受参数,则可省略形参。例如:

```
int current_power(); // current_power doesn't take an argument
```

如果你不希望函数返回一个结果,可将其返回类型设置为 void。例如:

```
void increase_power(int level); // increase_power doesn't return a value
```

这里, void 的意思是“不返回一个值”或“什么也不返回”。

在函数声明和定义中,你可以为参数命名也可以不命名,完全取决于你的需要。例如:

```
// search for s in vs;
// vs[hint] might be a good place to start the search
// return the index of a match; -1 indicates "not found"
int my_find(vector<string> vs, string s, int hint); // naming arguments
```

```
int my_find(vector<string>, string, int); // not naming arguments
```

在函数声明中,形参的名字不是必需的,只是对于编写好的注释很有益处。从编译器的角度看,第二个 my_find() 声明与第一个一样好:它包含所有调用 my_find() 所需的信息。

通常,我们会命名函数定义中的所有参数,例如:

```
int my_find(vector<string> vs, string s, int hint)
// search for s in vs starting at hint
{
    if (hint<0 || vs.size()-1<hint) hint = 0;
    for (int i = hint; i<vs.size(); ++i) // search starting from hint
        if (vs[i]==s) return i;
    if (0<hint) { // if we didn't find s search before hint
        for (int i = 0; i<hint; ++i)
            if (vs[i]==s) return i;
    }
    return -1;
}
```

参数 hint 使代码复杂了许多,但它的使用是基于这样一个假设: my_find() 的使用者粗略知道如何在一个向量中找到一个字符串,所以使用 hint 可以导致好的效果。但是,假设我们已经使用了 my_find() 一段时间,发现调用者很少能用好 hint,因此它实际上降低了性能。现在不再需要 hint 了,但是已有大量“外部”代码调用 my_find() 时使用了参数 hint。我们不想重写这些代码(或者因为是他人所写代码无法修改),因此我们不想更改 my_find() 的声明。替代方法是,我们不再使用最后一个参数(但在声明中保留它)。由于不再使用,所以可以不为其命名:

```
int my_find(vector<string> vs, string s, int) // 3rd argument unused
{
    for (int i = 0; i<vs.size(); ++i)
        if (vs[i]==s) return i;
    return -1;
}
```

你可在 Stroustrup 的《The C++ Programming Language》一书中和 ISO C++ 标准中找到变量和函数定义的完整语法。

8.5.2 返回一个值

我们可以用 return 语句从函数返回一个值:

```
T f() // f() returns a T
{
    V v;
```

```
// ...
return v;
}
```

```
T x = f();
```

这段代码中的返回值恰好就是我们用一个类型为 V 的值初始化一个类型为 T 的变量所得到的值：

```
V v;
// ...
T t(v);    // initialize t with v
```

也就是说，返回值可以看做初始化的另一种形式。如果函数声明中指定要返回值，则函数体内必须通过 return 返回一个值。否则，就会导致错误“直至函数末尾未返回值”：

```
double my_abs(int x)    // warning: buggy code
{
    if (x < 0)
        return -x;
    else if (x > 0)
        return x;
}    // error: no value returned if x is 0
```

实际上，编译器可能不会注意到我们“忘记了”`x == 0` 的情形。原则上它应该注意到，但很少有编译器如此聪明。对于复杂的函数，编译器完全可能无法知道你是否返回了一个值，因此编程中要小心。这里，“小心”的意思是，要切实保证对于函数的每种执行路径都有一条 return 语句或一个 error()。

由于历史原因，main() 是一个特例。执行到 main() 的末尾而未返回值，等价于返回 0，意思是“成功完成”程序。

在一个不返回值的函数中，我们可以调用无值的 return 语句从函数返回调用者。例如：

```
void print_until_s(const vector<string> v, const string quit)
{
    for(int i=0; i<v.size(); ++i) {
        if (v[i]==quit) return;
        cout << v[i] << '\n';
    }
}
```

如你所见，在一个 void 函数中直至末尾未返回值是合法的，这等价于一个无值的返回 return;。

8.5.3 传值参数

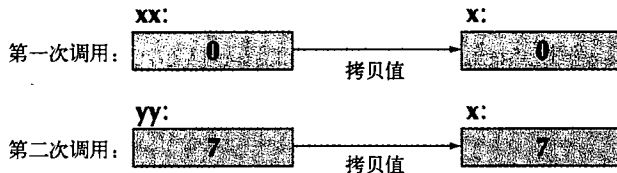
向函数传递参数最简单的方式是，将参数的值拷贝一份，交给函数。一个函数 f() 的参数实际上是 f() 中的局部变量，每次 f() 被调用时都会初始化。例如：

```
// pass-by-value (give the function a copy of the value passed)
int f(int x)
{
    x = x+1;    // give the local x a new value
    return x;
}

int main()
{
    int xx = 0;
    cout << f(xx) << endl;    // write: 1
    cout << xx << endl;    // write: 0; f() doesn't change xx

    int yy = 7;
    cout << f(yy) << endl;    // write: 8
    cout << yy << endl;    // write: 7; f() doesn't change yy
}
```

由于传递的是拷贝，因此 $f()$ 中的 $x = x + 1$ 不会改变两次调用时传递的变量 xx 和 yy 的值。下图可以说明传值参数的机制：



传值方式非常直接，其代价就是拷贝值的代价。

8.5.4 传常量引用参数

当传递占用存储空间小的值，如一个整数、一个双精度数或一个单词（参见 6.3.2 节）时，传值方式简单、直接、高效。但对于占用存储空间大的值，如一个图像（通常有几百位大小）、一个大表（比如几千个整数）或一个长字符串（比如几百个字符）时，又如何呢？在这种情况下，拷贝的代价就会非常高。我们不必为拷贝代价所困扰，但做不必要的工作就可能会很麻烦了，这意味着我们不能直接表达想要什么。例如，我们可能会编写下面这个函数来打印一个浮点数向量：

```

void print(vector<double> v)    // pass-by-value; appropriate?
{
    cout << "{ ";
    for (int i = 0; i < v.size(); ++i) {
        cout << v[i];
        if (i != v.size() - 1) cout << ", ";
    }
    cout << " }\n";
}

```

这个 `print()` 函数适用于所有规模的向量，例如：

```

void f(int x)
{
    vector<double> vd1(10);           // small vector
    vector<double> vd2(1000000);      // large vector
    vector<double> vd3(x);            // vector of some unknown size
    // ... fill vd1, vd2, vd3 with values ...
    print(vd1);
    print(vd2);
    print(vd3);
}

```

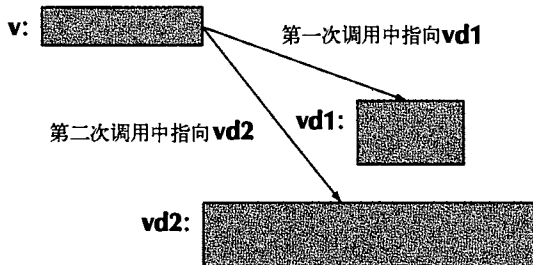
这段代码可以得到我们想要的结果，但首次调用 `print()` 需要拷贝 10 个双精度数（大概 80 个字节），第二次调用需要拷贝 100 万个双精度数（大概 8 兆字节），而第三次调用需要拷贝多少字节我们不知道。在此，我们必须问自己一个问题：“为什么我们要拷贝全部数据？”我们只不过是打印向量，而不是拷贝它们的元素。显然，必须要有一种方法，能使我们向函数传递一个变量，而不拷贝其值。一个类似的例子是，如果你被分派了一个任务——为图书馆中的书籍建立一个目录，馆长不会给你送去图书馆大楼和其内所有内容的一份拷贝，而只是把图书馆的地址发送给你，这样你就能到图书馆去浏览馆藏书籍。因此，我们需要某种方法，能将要打印的向量的“地址”而不是其拷贝传送给 `print()` 函数。这样的“地址”称为引用（reference），其使用方法如下：

```
void print(const vector<double>& v)    // pass-by-const-reference
{
    cout << " ";
    for (int i = 0; i < v.size(); ++i) {
        cout << v[i];
        if (i != v.size() - 1) cout << ", ";
    }
    cout << " \n";
}
```

符号 `&` 表示“引用”，而此处的 `const` 用来阻止 `print()` 无意中修改其参数。除了参数声明进行了修改外，代码所有其他部分都与传值方式的版本完全一样；唯一的改变在于 `print()` 现在是通过引用“提取到”参数的值，而非拷贝操作。注意短语“提取”（refer back），这种参数之所以称为引用，是因为它们“指向”定义于别处的对象（它们并不是对象本身）。我们可以像以前一样调用新版本的 `print()`：

```
void f(int x)
{
    vector<double> vd1(10);           // small vector
    vector<double> vd2(1000000);      // large vector
    vector<double> vd3(x);           // vector of some unknown size
    // ... fill vd1, vd2, vd3 with values ...
    print(vd1);
    print(vd2);
    print(vd3);
}
```

传常量引用方式如下图所示：



常量（`const`）引用的一个非常有用的特性是，我们不可能意外地修改传来的对象。例如，如果我们犯了一个愚蠢的错误，试图在 `print()` 中修改向量元素，编译器就会发现这个错误：

```
void print(const vector<double>& v)    // pass-by-const-reference
{
    // ...
    v[i] = 7;    // error: v is a const (is not mutable)
    // ...
}
```

传常量引用参数是一种有用的、常用的机制。请再次考虑 `my_find()` 函数（参见 8.5.1 节），它在一个字符串向量中搜索一个字符串，传值参数会导致不必要的拷贝代价：

```
int my_find(vector<string> vs, string s);    // pass-by-value: copy
```

如果向量中包含数千个字符串，即便在一台非常快的计算机上，你也能观察到拷贝所花费的时间。因此，我们可以通过将 `my_find()` 的参数改为传常量引用方式来改进它：

```
// pass-by-const-reference: no copy, read-only access
int my_find(const vector<string>& vs, const string& s);
```


8.5.5 传引用参数

但是，如果我们确实希望函数修改其参数，又该怎么办呢？有时，我们有充足的理由需要这么做。例如，我们可能需要一个 `init()` 函数为向量元素赋值：

```
void init(vector<double>& v)    // pass-by-reference
{
    for (int i = 0; i < v.size(); ++i) v[i] = i;
}

void g(int x)
{
    vector<double> vd1(10);      // small vector
    vector<double> vd2(1000000); // large vector
    vector<double> vd3(x);       // vector of some unknown size

    init(vd1);
    init(vd2);
    init(vd3);
}
```

这里，我们希望 `init()` 函数修改参数向量，因此我们没有使用传值参数（拷贝参数值），也没有使用传常量引用参数（不允许修改参数），只是将实际参数的“简单引用”传递给形参。

让我们从更技术化的角度来探讨一下引用。引用是这样一种语法机制，它允许用户为一个对象声明一个新的名字。例如，`int&` 是一个整型对象的引用，因此，我们可写出如下代码：

```
int i = 7;

int& r = i;    // r is a reference to i
r = 9;        // i becomes 9
i = 10;
cout << r << ' ' << i << '\n';    // write: 10 10
```



也就是说，任何对 `r` 的使用实际上使用的是 `i`。

引用的一个用途是作为简写形式。例如，我们可能用到如下二维向量：

```
vector< vector<double> > v;    // vector of vector of double
```

我们需要多次使用某个向量元素 `v[f(x)][g(x)]`。`v[f(x)][g(x)]` 是一个复杂的表达式，我们当然不愿意反复输入它。如果我们只是需要这个元素的值，那么可以声明下面这个变量：

```
double val = v[f(x)][g(y)];    // val is the value of v[f(x)][g(y)]
```

然后多次使用 `val` 即可。但如果我们既要到 `v[f(x)][g(x)]` 中读取值，又要向它写入值呢？这时，引用就派上用场了：

```
double& var = v[f(x)][g(y)];    // var is a reference to v[f(x)][g(y)]
```

现在，通过 `var`，我们既可以从 `v[f(x)][g(x)]` 中读取值，也可以向它写入值。例如：

```
var = var/2+sqrt(var);
```

引用的这一重要特性，即可以方便地作为某个对象的简写形式的特性，是引用能作为一种有用的参数传递方式的原因。例如：

```
// pass-by-reference (let the function refer back to the variable passed)
int f(int& x)
{
    x = x+1;
    return x;
}
```

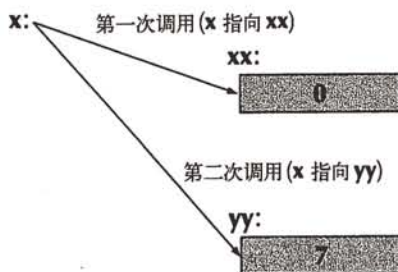
```

int main()
{
    int xx = 0;
    cout << f(xx) << endl;    // write: 1
    cout << xx << endl;      // write: 1; f() changed the value of xx

    int yy = 7;
    cout << f(yy) << endl;    // write: 8
    cout << yy << endl;      // write: 8; f() changed the value of yy
}

```

下图说明了上例中传引用方式参数传递的原理：



请将此例与 8.5.3 节中的类似实例进行比较。

传引用参数显然是一种非常强大的机制：在函数中我们可以直接操作任何以引用方式传递来的对象。例如，交换两个值是很多算法（例如排序）中非常重要的操作。利用引用，我们可以编写下面这样一个交换两个浮点数的函数：

```

void swap(double& d1, double& d2)
{
    double temp = d1;    // copy d1's value to temp
    d1 = d2;             // copy d2's value to d1
    d2 = temp;           // copy d1's old value to d2
}

int main()
{
    double x = 1;
    double y = 2;
    cout << "x == " << x << " y == " << y << "\n";    // write: x==1 y==2
    swap(x,y);
    cout << "x == " << x << " y == " << y << "\n";    // write: x==2 y==1
}

```

标准库提供了一个 `swap()` 函数，可以用来交换任何类型的值，只要该类型支持拷贝操作即可。因此，你不需要为每个类型编写自己的 `swap()` 函数。

8.5.6 传值与传引用的对比

如何在传值方式、传引用方式和传常量引用方式间进行选择呢？我们先来看第一个例子：

```

void f(int a, int& r, const int& cr)
{
    ++a;    // change the local a
    ++r;    // change the object referred to by r
    ++cr;    // error: cr is const
}

```

如果你希望改变被传递的对象的值，你应该使用非常量的引用：传值方式传来的是对象的拷贝；

而传常量引用方式不允许你修改对象的值。你可以试试下面的程序，观察三种参数传递方式的效果：

```
void g(int a, int& r, const int& cr)
{
    ++a;      // change the local a
    ++r;      // change the object referred to by r
    int x = cr; // read the object referred to by cr
}

int main()
{
    int x = 0;
    int y = 0;
    int z = 0;

    g(x,y,z);    // x==0; y==1; z==0
    g(1,2,3);    // error: reference argument r needs a variable to refer to
    g(1,y,3);    // OK: since cr is const we can pass a literal
}
```

如果你想改变通过引用方式传递过来的对象的值，你必须传递一个对象，而不能是一个常量。从技术上讲，整型文字常量 2 只是一个值(右值, rvalue)，而不是一个能保存值的对象。而这里函数 g() 的参数 r 需要的是一个左值(lvalue)，即可以出现在赋值号左边的内容。

注意，常量引用不需要一个左值，它可以像初始化和传值方式一样进行转换。在上面的代码中，当进行最后一次调用 g(1, y, 3) 时发生了什么呢？情况是这样的，编译器为函数 g() 的参数 cr 分配了一个整型变量，令 cr 指向它：

```
g(1,y,3); // means: int __compiler_generated = 3; g(1,y,__compiler_generated)
```

这种编译器生成的对象称为临时对象(temporary object)。

我们的根本原则是：

- 1) 使用传值方式传递非常小的对象。
- 2) 使用传常量引用方式传递你不需要修改的大对象。
- 3) 让函数返回一个值，而不是修改通过引用参数传递来的对象。
- 4) 只有迫不得已时才使用传引用方式。

这些原则会帮我们写出最简单、最不易出错而且最高效的代码。“非常小”的意思是一个或两个整型数、一个或两个双精度数或和它们差不多大小的对象。如果我们发现一个参数是以非常量引用方式传递的，我们必须假设被调用的函数会修改这个参数。

第三条规则表达的是，当你想用函数中改变一个变量的值时，实际上你还有另一种选择。考虑如下代码：

```
int incr1(int a) { return a+1; } // return the new value as the result
void incr2(int& a) { ++a; }    // modify object passed as reference

int x = 7;
x = incr1(x); // pretty obvious
incr2(x);    // pretty obscure
```

那么我们为什么还需要非常量引用传递方式呢？因为有时候这种参数传递方式是必要的

- 用于操作容器(比如向量)
- 用于改变多个对象的函数(函数只能有一个返回值)

例如：

```

void larger(vector<int>& v1, vector<int>& v2)
    // make each element in v1 the larger of the corresponding
    // elements in v1 and v2;
    // similarly, make each element of v2 the smaller
{
    if (v1.size()!=v2.size() error("larger(): different sizes");
    for (int i=0; i<v1.size(); ++i)
        if (v1[i]<v2[i])
            swap(v1[i],v2[i]);
}

void f()
{
    vector<int> vx;
    vector<int> vy;
    // read vx and vy from input
    larger(vx,vy);
    // ...
}

```

对于 larger() 这样的函数来说, 使用传引用参数是唯一合理的选择。

通常最好避免让函数修改多个对象。理论上, 总会有替代方法, 比如返回一个包含多个值的类对象。但是, 已经有大量程序使用了修改一个或多个参数的函数, 因此你很可能遇到这类程序。例如, 在 Fortran(大约 50 年中一直是用于数值计算的主要编程语言)中, 所有参数都是以引用方式传递的。很多数值计算程序直接借鉴了已有的 Fortran 程序, 调用了 Fortran 函数。这些代码通常使用传引用方式和传常量引用方式。

如果我们使用引用只是想避免拷贝操作, 那可以使用常量引用。这样, 当我们看到一个非常量引用参数时, 我们就可以假定这个函数改变了参数的值; 也就是说, 当我们看到一个非常量引用的参数传递时, 我们假定这个函数不仅是修改参数的值, 而是确实这么做了, 因此我们必须小心检查对函数的调用, 确保它按我们所期待的那样工作。

8.5.7 参数检查和转换

参数传递过程就是用函数调用中指定的实际参数初始化函数的形式参数的过程, 考虑如下代码:

```

void f(T x);
f(y);
T x=y;    // initialize x with y (see §8.2.2)

```

只要初始化语句 $T\ x = y$; 合法, 函数调用 $f(x)$ 就是合法的, 此时, 两个 x (初始化的变量和函数的参数) 会获得相同的值。例如:

```

void f(double);

void g(int y)
{
    f(y);
    double x(y);
}

```

注意, 用 y 初始化 x 时, 我们必须将一个整数转换为一个双精度数。在调用函数 $f()$ 时, 会进行同样的操作。 $f()$ 收到的双精度值与变量 x 中保存的值是一样的。

类型转换在一般情况下是很有用的, 但偶尔会带来奇怪的结果(参见 3.9.2 节)。因此, 我们对类型转换必须小心。例如, 如果一个函数要求一个整数, 那么向它传递一个双精度参数就不是一个好主意:

```

void ff(int);

void gg(double x)
{
    ff(x);    // how would you know if this makes sense?
}

```

如果你确实是想将一个双精度值截取为一个整数，请使用显式类型转换：

```

void ggg(double x)
{
    int x1 = x;    // truncate x
    int x2 = int(x);

    ff(x1);
    ff(x2);

    ff(x);        // truncate x
    ff(int(x));
}

```

使用显式类型转换的代码，其他程序员容易从中看出你的思路。

8.5.8 实现函数调用

当一个函数被调用时，计算机实际上做了什么呢？第6章和第7章中的函数 `expression()`、`term()` 和 `primary()` 可以很好地说明这一问题，除了一个细节：这些函数都不接受参数，因此我们无法用它们解释参数是如何传递的。但是，请等一下！这些函数必然是获取一些输入的，否则它们不可能做任何有用的事情。实际上它们接受了一个隐含的参数：它们使用了一个称为 `ts` 的 `Token_stream` 对象来获得输入，而 `ts` 是一个全局变量。我们可以改进这些函数，让它们接受一个 `Token_stream&` 类型的参数。因此本节中这几个函数都被增加了一个 `Token_stream&` 参数，而所有与函数调用实现不相关的内容都被去掉了。

首先，函数 `expression()` 非常简单，它有一个参数(`ts`)和两个局部变量(`left` 和 `t`)：

```

double expression(Token_stream& ts)
{
    double left = term(ts);
    Token t = ts.get();
    // ...
}

```

其次，函数 `term()` 与 `expression()` 非常类似，只是多了一个额外的局部变量(`d`)，用来保存除法运算的除数。

```

double term(Token_stream& ts)
{
    double left = primary(ts);
    Token t = ts.get();
    // ...
    case '/':
    {
        double d = primary(ts);
        // ...
    }
    // ...
}

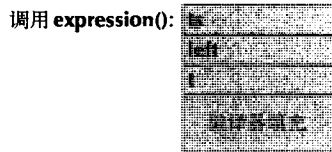
```

第三，函数 `primary()` 与 `term()` 很类似，只是多了一个局部变量 `left`：

```
double primary(Token_stream& ts)
{
    Token t = ts.get ();
    switch (t.kind) {
    case '/':
        {    double d = expression(ts);
            // ...
        }
        // ...
    }
}
```

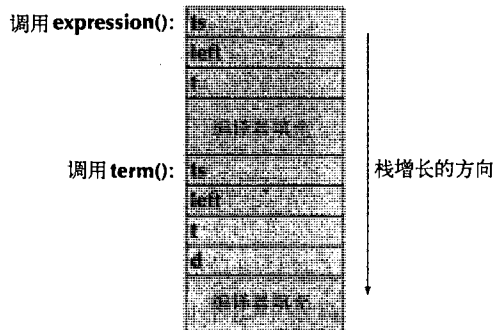
现在这些函数已经不再使用任何“鬼鬼祟祟的全局变量”了，用来说明函数调用机制已经非常理想了：它们都有一个参数，都有局部变量，而且它们相互调用。你可能希望有机会重新回顾一下完整的 `expression()`、`term()` 和 `primary()` 是什么样，但与函数调用相关的特性这里都已经给出了。

当一个函数被调用时，编译器分配一个数据结构，保存所有参数和局部变量的拷贝。例如，当 `expression()` 第一次被调用时，编译器会创建如下数据结构：



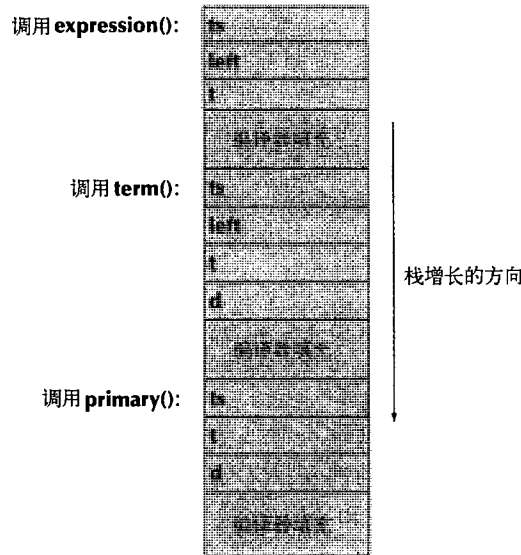
在“编译器填充”部分，不同编译器填入的内容不同，但基本上是函数返回到调用者以及返回一个值给调用者所需的信息。这样的数据结构称为函数活动记录，每个函数的活动记录都有自己特有的详细布局。注意，从编译器的角度看，一个参数只是另一个局部变量而已。

到目前为止，一切都很好，现在 `expression()` 调用 `term()`，编译器会为 `term()` 的这次调用创建相应的活动记录：

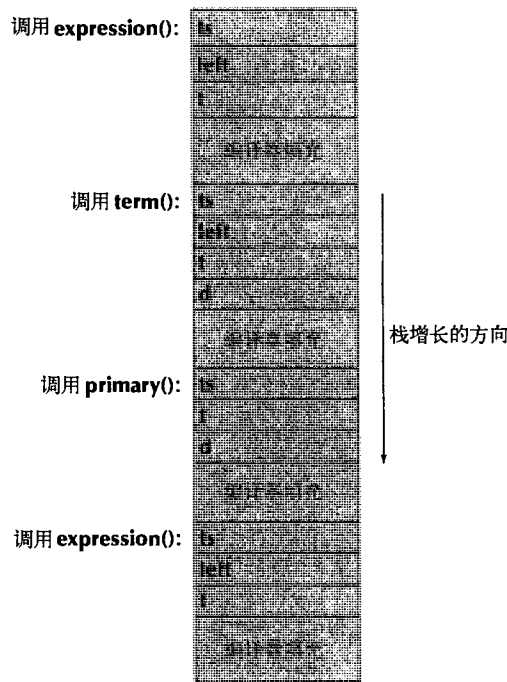


我们注意到 `term()` 需要保存有一个额外的变量 `d`，因此在调用中编译器为它分配了存储空间，即使程序中可能永远也不使用它。这没有问题，对于合理的函数（比如本书中我们直接或间接使用的所有函数）来说，创建一个函数活动记录的运行时代价不依赖于它有多大。局部变量 `d` 只有当我们执行 `case '/'` 时才会被初始化。

现在 `term()` 调用 `primary()`，编译器会创建如下活动记录：

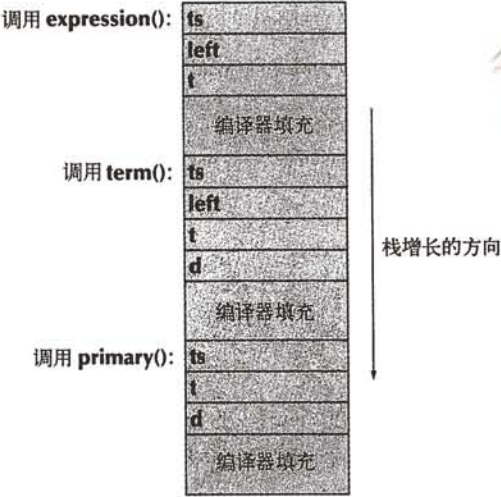


这么叙述下去看来有些啰嗦了，但现在 `primary()` 调用 `expression()`：

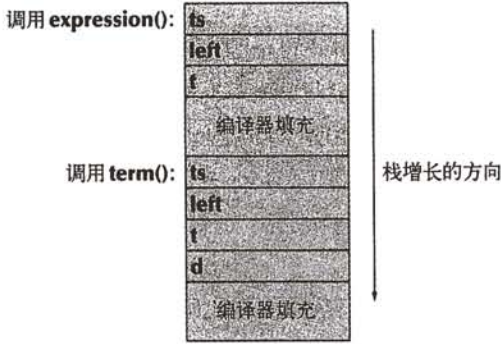


编译器为 `expression()` 的这次调用创建了它自己的活动记录，与第一次 `expression()` 调用的活动记录是不同的。这样 `left` 和 `t` 在两次调用中是不同的，这是一种很好的处理方式，否则我们将处于糟糕的境地。一个函数如果直接或间接（在本例中）调用自身的话，我们称之为递归（recursive）。如你所见，正是因为有了上述函数调用和返回的实现技术，递归函数才得以成立（反之亦然）。

因此，每当我们调用函数时，活动记录栈（stack of activation record）——通常就称 `0` 为栈（stack）生长出一个记录。反过来，当函数返回时，其记录就不再有用。例如，当最后一个 `expression()` 调用返回 `primary()` 时，栈将复原为下图：



当 `primary()` 返回 `term()` 时，栈回到如下图所示：



依此类推。这里使用的栈，也称为调用栈 (call stack)，是一种只在一端增长和缩减的数据结构，其增长和缩减的规则是：后进先出。

请记住不同 C++ 编译器实现和使用调用栈的细节是不同的，但基本的原理就大致如上文所述。为了使用函数，你需要知道函数调用是如何实现的吗？当然不需要，你在前面学习的使用函数的知识已经足够多、足够好了，学习本小节关于函数调用实现方面的内容不是必需的，但很多程序员还是想知道这方面的知识，而且很多程序员使用诸如“活动记录”、“调用栈”之类的术语，所以了解一下这方面的内容还是有好处的。

8.6 求值顺序

一个程序的求值，或称为程序的执行，就是按照语言的规则逐条运行程序中的语句。当这条“执行线索”到达一个变量定义时，变量就会被创建；也就是说，编译器会为它分配内存空间，并对其进行初始化。当变量退出其作用域时，将会被销毁，即原则上它所指向的对象会被删除，编译器可把它原来占用的内存用做他用。例如：

```
string program_name = "silly";  
vector<string> v;           // v is global  
  
void f()  
{
```

```

string s;                                // s is local to f
while (cin>>s && s!="quit") {
    string stripped;                     // stripped is local to the loop
    string not_letters;
    for (int i=0; i<s.size(); ++i)      // i has statement scope
        if (isalpha(s[i]))
            stripped += s[i];
        else
            not_letters += s[i];
    v.push_back(stripped);
    // ...
}
// ...
}

```

像 `program_name` 和 `v` 这样的全局变量，在 `main()` 的第一条语句执行之前就会被初始化。其生存期直至程序结束，随后会被销毁。它们创建的顺序与定义的顺序相符（`program_name` 先于 `v` 创建），而销毁则按相反的次序（即 `v` 先于 `program_name` 被销毁）。

当有代码调用 `f()` 时，首先会创建 `s`，并将其初始化为空字符串，`s` 的生命期会持续到从 `f()` 返回的时刻。

每次进入 `while` 循环的循环体时，`stripped` 和 `not_letters` 两个变量会被创建。由于 `stripped` 的定义在 `not_letters` 之前，因此它也先被创建。两个变量的生存期都是到本次循环步的结束为止，在循环条件重新求值之前被销毁，销毁顺序与创建顺序相反（即 `not_letters` 先于 `stripped` 被销毁）。因此，如果我们在遇到字符串“quit”之前读入 10 个其他的字符串，`stripped` 和 `not_letters` 将会被创建和销毁 10 次。

每次到达 `for` 循环时，`i` 会被创建。每次退出 `for` 循环时，到达语句 `v.push_back(stripped);` 前，`i` 会被销毁。

请注意，编译器是聪明的家伙，它们获准优化代码，只要得到的结果与我们的目标相符即可。特别是，编译器在分配/释放内存方面很聪明，不会不必要地频繁分配/释放内存。

8.6.1 表达式求值

表达式中子表达式求值顺序所遵循的规则，是按优化编译器的需求设计的，而不是为了方便程序员。这很不幸，但不管怎样你应该避免复杂的表达式，有一条简单的原则可以帮你远离麻烦：如果你在表达式中改变一个变量的值，不要在同一个表达式中再读或写这个变量。例如：

```

v[i] = ++i;                            // don't: undefined order of evaluation
v[++i] = i;                            // don't: undefined order of evaluation
int x = ++i + ++i;                     // don't: undefined order of evaluation
cout << ++i << ' ' << i << '\n';      // don't: undefined order of evaluation
f(++i, ++i);                          // don't: undefined order of evaluation

```

不幸的是，如果你写出这样有问题的代码，并不是所有编译器都能给出警告。这段代码的问题在于，如果你将代码迁移到另外一台计算机，使用另一个编译器，或者改变编译器优化设置，运行结果不保证一致。不同的编译器确实会对这段代码得到不同的结果，所以不要这么编写程序。

特别要注意的是，`=`（赋值符）在表达式中只是又一种运算符而已，并没有特殊的地位，因此不能保证赋值符左边的子表达式在右边的子表达式之前求值。这就是为什么 `v[++i] = i` 的结果是不确定的。

8.6.2 全局初始化

在同一个编译单元中的全局变量（以及名字空间变量，参见 8.7 节）按它们出现的顺序被初始化。例如：

```
// file f1.cpp
int x1 = 1;
int y1 = x1+2;    // y1 becomes 3
```

逻辑上,这几个变量的初始化在 `main()` 中的代码执行前发生。

除非是在一些非常特殊的情况下,否则一般来说使用全局变量不是一个好主意。我们已经提到过,程序员没有有效的方法获知程序的哪个部分读/写了一个全局变量(参见 8.4 节)。另一个问题是,在不同编译单元中的全局变量的初始化顺序是不确定的。例如:

```
// file f2.cpp
extern int y1;
int y2 = y1+2;    // y2 becomes 2 or 5
```

这段代码存在这样几个问题:使用了全局变量;为全局变量起了很短的名字;对全局变量使用了复杂的初始化。如果文件 `f1.cpp` 中的全局变量先于文件 `f2.cpp` 中的全局变量被初始化,那么 `y2` 的初值为 5(这可能是程序员本来所期望的,也是合理的)。但是,如果文件 `f2.cpp` 中的全局变量先于文件 `f1.cpp` 中的全局变量被初始化,`y2` 的初值将为 2(因为分配给全局变量的内存空间,在变量的复杂初始化前被置为 0)。请避免使用这种代码,并且对复杂的初始化保持足够的警惕,任何不是简单常量表达式的初始化都可以认为是复杂的。

但如果确实需要一个全局变量(或常量),而且需要对它进行复杂的初始化,你又该怎么做呢?一个看起来有道理的例子是,一个用于商务事务的库需要一个 `Date` 类型的对象,我们想初始化这个对象:

```
const Date default_date(1970,1,1);    // the default date is January 1, 1970
```

如何知道 `default_date` 在初始化之前从未使用过呢?原则上我们不可能知道,因此我们不应该写出这样的代码。一种常用的技术是编写一个函数,返回我们需要的值。例如:

```
const Date default_date()    // return the default Date
{
    return Date(1970,1,1);
}
```

每当我们调用 `default_date()` 函数,它都会为我们创建一个 `Date` 对象。一般情况下,这种技术已经足够好,但如果需要频繁调用 `default_date()`,而且构造 `Date` 对象代价较高的话,我们更倾向于只构造它一次。可以这样做:

```
const Date& default_date()
{
    static const Date dd(1970,1,1);    // initialize dd first time we get here
    return dd;
}
```

一个静态的局部变量只在函数首次调用时才被初始化(被创建)。注意,这里返回一个引用,因此消除了不必要的对象拷贝。特别地,我们返回了一个常量引用,可以防止调用者无意中改变对象的值。本书之前对于如何传递参数的讨论,对返回值也是适用的(参见 8.5.6 节)。

8.7 名字空间

在函数中我们用程序块来组织代码(参见 8.4 节)。我们用类来将函数、数据和类型组织到一个类型中(参见第 9 章)。函数和类都为我们做了如下工作:

- 允许我们定义大量实体,而无需担心它们的名字与程序中其他实体的名字有冲突。
- 为我们提供了一个名字,用来访问我们定义的东西。

至此,我们还缺少一种技术,无需定义一个类型就能将类、函数、数据和类型组织成一个可识别

的命名实体。实现这种声明分组功能的 C++ 机制就是名字空间 (namespace)。例如, 我们希望提供一个包含类 Color、Shape、Line、Function 和 Text 的绘图库 (参见第 13 章):

```
namespace Graph_lib {
    struct Color { /* ... */ };
    struct Shape { /* ... */ };
    struct Line : Shape { /* ... */ };
    struct Function : Shape { /* ... */ };
    struct Text : Shape { /* ... */ };
    // ...
    int gui_main() { /* ... */ }
}
```

很可能其他人也使用了这些名字, 但没有关系。你可以定义名为 Text 的实体, 但与我们的 Text 没有冲突。Graph_lib::Text 是我们定义的类, 而你的 Text 与之不同。唯一可能有问题的情况就是, 你也定义了一个名为 Graph_lib 的类或名字空间, 它也包含一个名为 Text 的成员。Graph_lib 这个名字有点丑, 我们选择它的原因是, “漂亮且清晰” 的名字 Graphics 有很大可能已经被别人用过了。

假设你的 Text 是一个文字处理库的一部分。我们用来将绘图功能组织到名字空间 Graph_lib 中的思想, 也可用来将你的文字处理功能组织到一个叫其他名字 (比如 TextLib) 的名字空间:

```
namespace TextLib {
    class Text { /* ... */ };
    class Glyph { /* ... */ };
    class Line { /* ... */ };
    // ...
}
```

如果我们定义的这两个名字空间都是全局的, 我们可能会陷入真正的麻烦之中。假如有人同时使用我们的这两个库, 就可能真的遇到名字冲突, 如 Text 和 Line。糟糕的是, 如果我们的两个库都有用户在使用, 我们就无法通过修改 Line 和 Text 这些名字来避免冲突, 否则用户程序也必须修改。为了解决这一问题, 我们可以使用名字空间, 即我们的 Text 用 Graph_lib::Text 表示, 你的 Text 用 TextLib::Text。这种将一个名字空间的名字 (或一个类名) 和一个成员名用 :: 组合成的名字称为全限定名 (fully qualified name)。

8.7.1 using 声明和 using 指令

使用全限定名太繁琐了。例如, C++ 标准库的功能都定义在 std 名字空间中, 因此可以按如下方式使用:

```
#include<string>           // get the string library
#include<iostream>         // get the iostream library

int main()
{
    std::string name;
    std::cout << "Please enter your first name\n";
    std::cin >> name;
    std::cout << "Hello, " << name << "\n";
}
```

我们已经见过标准库中的 string 和 cout 无数次了, 我们真不希望必须用 “正确的” 全限定名 std::string 和 std::cout 才能访问它们。如果有这么一种方法, 能实现 “当我说 string, 我的意思是 std::string”、“当我说 cout, 我的意思是 std::cout” 等, 那就好了, 就像下面这样:

```
using std::string;         // string means std::string
using std::cout;           // cout means std::cout
// ...
```

这种语法结构称为 using 声明，它与我们常用的人名简称类似：你可以简单地用“Greg”来代表 Greg Hansen，只要屋子里没有其他叫 Greg 的人就没问题。

有时，我们希望可以有一种更强的“简称”用来引用名字空间中的名字：“如果你在本作用域中没有发现某个名字的声明，那么就在 std 中寻找它”。使用 using 指令可以达到这个目的：

```
using namespace std; // make names from std directly accessible
```

于是我们得到了下面这种常用的程序风格：

```
#include<string>           // get the string library
#include<iostream>         // get the iostream library
using namespace std;       // make names from std directly accessible

int main()
{
    string name;
    cout << "Please enter your first name\n";
    cin >> name;
    cout << "Hello, " << name << '\n';
}
```

其中 cin 表示 std::cin，string 表示 std::string，依此类推。只要你使用 std_lib_facilities.h，你就不需要担心标准库头文件和 std 名字空间了。

一个一般性的原则是，除非是 std 这种在某个应用领域中大家已经非常熟悉的命名空间，否则最好不要使用 using 指令。过度使用 using 指令带来的问题是，你已经记不清每个名字来自哪里，结果就是你又陷入名字冲突之中。显式使用全限定名，或者使用 using 声明就不存在这个问题。因此，将一个 using 指令放在头文件中是一个非常坏的习惯，因为用户就无法避免上述问题。然而，为了简化初学者编写程序，我们确实在 std_lib_facilities.h 中为 std 放置了一个 using 指令，因此我们可以像下面代码这样来写程序：

```
#include "std_lib_facilities.h"

int main()
{
    string name;
    cout << "Please enter your first name\n";
    cin >> name;
    cout << "Hello, " << name << '\n';
}
```

对于除 std 之外的名字空间，我们保证不会这样做。



简单练习

1. 创建三个文件：my.h、my.cpp 和 use.cpp。头文件 my.h 包含：

```
extern int foo;
void print_foo();
void print(int);
```

源文件 my.cpp 包含 my.h 和 std_lib_facilities.h，定义 print_foo()，此函数用 cout 来打印 foo 的值，my.cpp 中还应定义函数 print(int i)，用 cout 打印 i 的值。

源文件 use.cpp 包含 my.h，定义主函数 main()，主函数中将 foo 的值置为 7，用 print_foo() 打印它，并用 print() 打印整型值 99。注意 use.cpp 并不包含 std_lib_facilities.h，因为它并不直接使用标准库中的功能。

编译并运行这些文件。在 Windows 平台上，你需要将 use.cpp 和 my.cpp 放在同一个项目中，并在 use.cpp 中使用 { char cc; cin >> cc; } 来看到输出结果。

2. 编写三个函数 swap_v(int, int)、swap_r(int&, int&) 和 swap_cr(const int&, const int&)。每个函数有如下

函数体:

```
{ int temp; temp = a, a=b; b=temp; }
```

其中 a 和 b 是参数名。

尝试用如下代码调用这三个函数:

```
int x = 7;
int y = 9;
swap_?(x,y); // replace ? by v, r, or cr
swap_?(7,9);
const int cx = 7;
const int cy = 9;
swap_?(cx,cy);
swap_?(7.7,9.9);
double dx = 7.7;
double dy = 9.9;
swap_?(dx,dy);
swap_?(dx,dy);
```

哪个调用能编译通过? 为什么? 对每个编译通过的 swap 调用, 在其调用过后打印参数的值, 检查参数值是否真正被交换了。如果你不理解得到的结果, 查阅 8.6 节。

3. 编写一个程序, 由一个文件组成, 其中包含三个名字空间 X、Y 和 Z, 使得如下 main() 函数能正确运行:

```
int main()
{
    X::var = 7;
    X::print();    // print X's var
    using namespace Y;
    var = 9;
    print();       // print Y's var
    {
        using Z::var;
        using Z::print;
        var = 11;
        print();   // print Z's var
    }
    print();       // print Y's var
    X::print();    // print X's var
}
```

每个名字空间需定义一个名为 var 的变量和一个名为 print() 的函数, 该函数用 cout 输出恰当的 var 值。



思考题

1. 声明和定义有何区别?
2. 如何从语法上区分函数声明和函数定义?
3. 如何从语法上区分变量声明和变量定义?
4. 对于第 6 章的计算器程序中的函数, 为什么不先声明就无法使用?
5. int a; 是一个定义, 还是只是一个声明?
6. 为什么说在变量声明时对其初始化是一个好的编程风格?
7. 一个函数声明可以包含哪些内容?
8. 恰当使用缩进有什么好处?
9. 头文件的用处是什么?
10. 什么是声明的作用域?
11. 有几种作用域? 请各举一例。
12. 类作用域和局部作用域有何区别?
13. 为什么应该尽量少用全局变量?
14. 传值和传引用有何区别?
15. 传引用和传常量引用有何区别?

16. 什么是 swap()?
17. 定义一个函数, 它带有一个 `vector<double>` 类型的传值参数, 这样做好吗?
18. 给出一个求值顺序不确定的例子, 并说明为什么求值顺序不确定是一个问题。
19. `x&&y` 和 `x||y` 分别表示什么?
20. 下面哪些语法结构符合 C++ 标准: 函数中的函数、类中的函数、类中的类、函数中的类。
21. 一个活动记录内都包含什么内容?
22. 什么是调用栈? 为什么需要调用栈?
23. 名字空间在作用是什么?
24. 名字空间和类有何区别?
25. using 声明是什么?
26. 为什么应该避免在头文件中使用 using 指令?
27. 命名空间 std 是什么?



术语

活动记录	函数定义	传值	实参	全局作用域	递归
参数传递	头文件	return	调用栈	初始化程序	返回值
类作用域	局部作用域	作用域	const	namespace	语句作用域
声明	名字空间作用域	技术细节	定义	嵌套语句块	未定义标识符
extern	形参	using 声明	前置声明	传常量引用	using 指令
函数	传引用				



习题

1. 修改第7章的计算器程序, 将输入流作为一个显式参数(如8.5.8节所示)。并为 `Token_stream` 设计接受 `istream&` 参数的构造函数, 这样, 当我们解决了如何使用自己的输入流时(如关联到一个文件), 就可以将之用于计算器程序。
2. 编写一个函数 `print()`, 将一个整型向量输出到 `cout`。此函数接受两个参数: 一个字符串(用于“标记”输出)和一个向量。
3. 创建一个斐波那契数的向量, 并用习题2中的函数输出这个向量。编写函数 `fibonacci(x, y, v, n)` 来创建向量, 其中 `x`、`y` 是 `int` 型, `v` 是 `vector<int>` 类型空向量, `n` 是要放入 `v` 的元素数目, 将 `v[0]` 和 `v[1]` 分别设置为 `x` 和 `y`。斐波那契数列是这样一个整型序列, 其中每个元素都是前面两个元素之和。例如, 以1和2开始, 可以得到斐波那契数列1、2、3、5、8、13、21、…。你设计的 `fibonacci()` 函数应该以参数 `x` 和 `y` 作为开始, 生成这样的斐波那契数列。
4. 计算机中 `int` 型对象能保存的整数的值是有上限的。使用 `fibonacci()` 函数求这个上限的近似值。
5. 编写两个函数, 反转一个 `vector<int>` 类型向量中的元素顺序。例如, 将1、3、5、7、9转换为9、7、5、3、1。第一个反转函数生成一个新向量, 其中元素为原向量的逆序, 而原向量内容不变。另一个反转函数不使用任何其他向量, 直接在原向量中反转元素顺序(提示: 用 `swap`)。
6. 对 `vector<string>` 类型向量, 重做习题5。
7. 读入5个名字, 存入一个 `vector<string>` 型向量 `name`, 然后提示用户输入这些人的年龄, 存入一个 `vector<double>` 型向量 `age`。然后打印5对 `(name[i], age[i])`。然后对名字排序(`sort(name.begin(), name.end())`), 并输出 `(name[i], age[i])` 对。此题的难点在于, 如何使 `age` 向量中元素的次序与已排序的 `name` 向量中的元素匹配。提示: 在排序 `name` 之前, 将它复制一份, 在排序之后, 利用此副本生成顺序正确的 `age` 副本。完成后, 重做此题, 使之能处理任意数目的姓名和年龄。
8. 编写一个简单函数 `randint()`, 它生成一个 `[0:MAXINT]` 之间的伪随机数。提示: 参考 Knuth 的《The Art of Computer Programming》第2版。
9. 编写一个函数 `rand_in_range(int a, int b)`, 它使用习题8的函数 `randint()` 生成 `[a:b]` 之间的一个伪随机数。注意: 此函数对于简单的游戏程序非常有用。

10. 编写一个函数，接受两个 `vector<double>` 型参数 `price` 和 `weight`，计算出一个值（“指数”）——所有 `price[i] * weight[i]` 之和。注意，我们必须保证 `weight.size() <= price.size()`。
11. 编写一个函数 `maxv()`，返回一个 `vector` 参数中的最大元素。
12. 编写一个函数，接受一个 `vector` 参数，找到最小和最大的元素，并计算均值和中位值。不要使用全局变量。或者返回一个 `struct`，包含所有计算结果；或者通过引用参数将所有计算结果返回。这两种返回多个结果的方法中，你更倾向于哪个？为什么？
13. 改进 8.5.2 节中的 `print_until_s()`，并测试它。什么样的测试用例集能更好地测试此程序？请解释原因。然后，编写一个 `print_until_ss()` 函数，它会一直打印，直至第二次看到它的 `quit` 参数。
14. 编写一个函数，接受一个 `vector<string>` 参数，返回一个 `vector<int>`，其每个元素值是对应字符串的长度。此函数还找出最长和最短的字符串，以及字典序第一个和最后一个字符串。为了完成这些工作，你需要用多少个独立的函数？为什么？
15. 能声明一个非引用的常量参数吗（如 `void f(const int);`）？这种参数传递方式意味着什么？为什么我们可能需要这种传参方式？为什么我们不应该经常使用这种方式？编写几个小程序来尝试这种传参方式，观察效果。



附录

我们可以将本章（以及第 9 章）的大部分内容放入附录。然而，你需要了解本章介绍的大部分 C++ 功能，以便学习本书的第二部分。用这些 C++ 功能可以很快地解决你遇到的很多问题，而这些问题是你承担的一些简单程序设计项目中所必须解决的。因此，为了节约时间，减少混淆，本章系统地介绍了这些内容，而不是让读者去“随机”地访问手册和附录。

第9章 类相关的技术细节

“记住，做事情要花费时间。”

——Piet Hein

在本章中，我们继续关注主要的程序设计工具——C++ 语言。本章主要介绍与用户自定义类型相关的语言技术细节，即类和枚举相关的内容。这些语言特性，大部分是以逐步改进一个 `Date` 类型的方式来介绍的。采用这种方式，我们还可以顺便介绍一些有用的类设计技术。

9.1 用户自定义类型

C++ 语言提供了一些内置类型，如 `char`、`int` 和 `double` (参见附录 A.8)。对于一个类型，如果编译器无须借助程序员在源码中提供的任何声明，就知道如何表示这种类型的对象以及可以对它进行什么样的运算 (如 `+` 和 `*`)，我们就称这种类型是内置的。

非内置的类型称为用户自定义类型 (user-defined type, UDT)。用户自定义类型包括每个 ISO 标准 C++ 实现都提供给程序员的标准库类型，如 `string`、`vector` 和 `ostream` (参见第 10 章)，也可以是我们为自己创建的类型，如 `Token` 和 `Token_stream` (参见 6.5 节和 6.6 节)。一旦我们掌握了足够多的必要的语言功能，我们就可以创建图形类型，如 `Shape`、`Line` 和 `Text` (参见第 13 章)。标准库类型很大程度上可以和内置类型一样看做语言的一部分，但我们还是将它们看做用户自定义类型，因为它们和我们自己创建的类型使用了同样的语言功能和技术；标准库的开发者并没有什么特权或者语言工具，是你我所不具备的。与内置类型相似，大多数用户自定义类型提供运算。例如，`vector` 有 `[]` 和 `size()` 运算 (参见 4.6.1 节和附录 B.4.8)，`ostream` 有 `<<`，`Token_stream` 有 `get()` (参见 6.8 节)，`Shape` 有 `add(Point)` 和 `set_color()` (参见 14.2 节)。

我们为什么要创建自定义类型呢？原因在于编译器不知道我们想在程序中使用的所有类型。它也不可能知道，因为有用的类型实在太多了——没有语言设计者或者编译器开发者能预知所有类型。我们每天都在创建新的类型。为什么？类型又有什么用？通过类型，我们可以在代码中直接、有效地表达我们的思想。当编写代码时，理想情况就是我们能在代码中直接表达我们的思想，这样我们自己、同事以及编译器就能理解代码的含义。当我们想进行算术运算时，`int` 类型会起到很大作用；当我们想处理文本时，`string` 类型会带来很大帮助；当我们想处理计算器的输入时，`Token` 和 `Token_stream` 会起到很大作用。这些类型带来的帮助体现在两个方面：

- 表示：一个类型“知道”如何表示对象中的数据。
- 运算：一个类型“知道”可以对对象进行什么运算。

很多编程想法都表现为这种模式：“某些东西”由表示它当前值的一些数据 (有时也称为当前状态)，及一组可以应用其上的运算组成。考虑一个计算机文件、一个网页、一个烤面包机、一台 CD 播放机、一个咖啡杯、一个汽车引擎、一部手机或是一本电话号码簿；每个对象都可以用一些数据描述，并且支持一组固定的、数量或多或少的标准操作。在每个例子中，操作的结果依赖于对象的数据——“当前状态”。

因此，我们希望在代码中将这样一个“想法”或“概念”表示为一个数据结构加上一组函数。

问题是“如何准确表示”。本章介绍一些在 C++ 中表述概念的一些基本方法，以及涉及的一些语言的技术细节。

C++ 提供了两类用户自定义数据类型：类和枚举。类是到目前为止最常用的，也是最重要的概念描述机制，因此我们首先把注意力放在类上。类能够在程序中直接地表达概念。一个类是一个(用户自定义)类型，它指出这种类型的对象如何表示，如何创建，如何使用，以及如何销毁(参见 17.5 节)。如果你将某些东西作为一个单独的实体来考虑，那么你可能就应该在程序中定义一个类来表示“这个东西”。这方面的例子有向量、矩阵、设备驱动、屏幕上的图片、对话框、图形、窗口、温度计读数以及时钟等。

在 C++ 中(以及大多数现代程序设计语言中)，类是构造大型程序的关键的基本组成部分，对小程序也同样非常有用，这一点我们已经在计算器程序中看到了(参见第 6 章和第 7 章)。

9.2 类和成员

一个类就是一个用户自定义类型，由一些内置类型和其他用户自定义类型的对象以及一些函数组成。这些用来定义类的组成部分称为成员(member)。一个类可以有 0 个或多个成员，例如：

```
class X {
public:
    int m;           // data member
    int mf(int v) { int old = m; m=v; return old; } // function member
};
```

成员可以有多种类别，大多数要么是数据成员，定义了类对象的表示方法，要么是函数成员，提供类对象之上的运算。类成员的访问使用这种符号：对象.成员。例如

```
X var;           // var is a variable of type X
var.m = 7;       // assign to var's data member m
int x = var.mf(9); // call var's member function mf()
```

你可以把 var.m 读做“var 的 m”，大多数人读做“var 点 m”或者“var 的 m”。一个成员的类型决定我们可以对它进行什么运算。例如，我们可以读/写一个 int 成员，可以调用一个函数成员，等等。

9.3 接口和实现

我们通常把一个类看做一个接口加上一个实现。接口是类声明的一部分，用户可以直接访问它。实现是类声明的另一部分，用户只能通过接口间接访问它。公共的接口用标号 public: 标识，实现用标号 private: 标识。你可以将一个类声明理解为如下形式：

```
class X { // this class's name is X
public:
    // public members:
    // - the interface to users (accessible by all)
    // functions
    // types
    // data (often best kept private)
private:
    // private members:
    // - the implementation details (used by members of this class only)
    // functions
    // types
    // data
};
```

类成员默认是私有的，也就是说，如以下代码所示：

```
class X {
    int mf(int);
    // ...
};
```

等价于

```
class X {
private:
    int mf(int);
    // ...
};
```

因此, 下面的代码是错误的

```
X x;           // variable x of type X
int y = x.mf(); // error: mf is private (i.e., inaccessible)
```

用户不能直接访问一个私有成员, 应通过一个公有函数来访问, 例如:

```
class X {
    int m;
    int mf(int);
public:
    int f(int i) { m=i; return mf(i); }
};

X x;
int y = x.f(2);
```

我们用私有和公有之间的差别来描述接口(类的用户视图)和实现细节(类的实现者视图)之间的重要区别。下面我们会逐步给出解释和大量实例, 在此我们仅仅指出: 如果类只包含数据, 接口和实现间的区别没有什么意义。C++ 提供了一种很有用的简化的功能, 可用来描述没有私有实现细节的类。这种语法功能就是结构, 一个结构就是一个成员默认为公有属性的类:

```
struct X {
    int m;
    // ...
};
```

意味着

```
class X {
public:
    int m;
    // ...
};
```

结构主要用于成员可以取任意值的数据结构, 即我们不能定义任何有意义的不变式(参见 9.4.3 节)。

9.4 演化一个类

下面, 让我们展示如何、以及为什么, 将一个简单数据结构逐步演化为一个具有私有实现细节和支持函数的类, 我们通过这些内容来说明支持类的语言功能和使用类的基本技术。我们对这些内容的介绍借助于一个非常普通的问题——如何在程序中表示日期(如 1954 年 8 月 14 日)。很显然, 很多程序都需要日期, 如商业事务程序、天气数据程序、日程表程序、工作记录、库存管理程序等。唯一的问题是, 我们如何才能表示它。

9.4.1 结构和函数

我们如何才能表示一个日期呢? 当我们提出这个问题时, 很多人会回答: “年、月、日, 这样表示如何?” 这不是唯一的答案, 也不总是最好的答案, 但目前对我们来说够用了, 这也是我们将

要采用的做法。我们的第一个方案是一个简单的结构：

```
// simple Date (too simple?)
struct Date {
    int y; // year
    int m; // month in year
    int d; // day of month
};
```

Date today; // a Date variable (a named object)

一个 Date 对象(如 today)由三个整型简单构成,如下图所示。这个 Date 结构不存在任何关联的隐藏数据结构,能“变出戏法”——而且本章中 Date 的任何一个版本也都是这样。

	Date:
y:	2005
m:	12
d:	24

现在已经有了表示日期的 Date,我们可以对它进行什么操作呢?实际上我们能做任何操作,因为我们可以访问 today(以及任何其他 Date 对象)的成员,可以按我们的意愿读写它们。困难在于事情确实不是那么方便,我们想对一个 Date 对象做任何事,都必须通过读写其成员的方式来进行,例如:

```
// set today to December 24, 2005
today.y = 2005;
today.m = 24;
today.d = 12;
```

这样编写程序冗长乏味,而且容易出错。你能看出上面代码中的错误吗?实际上,任何冗长乏味的东西都容易出错!例如,下面代码有任何意义吗?

```
Date x;
x.y = -3;
x.m = 13;
x.d = 32;
```

很有可能是没有意义的,而且没有人会这么写程序。再考虑下面的程序:

```
Date y;
y.y = 2000;
y.m = 2;
y.d = 29;
```

看起来比上一段程序有意义得多,但 2000 年是闰年吗?你确定?

较好的方法是设计一些辅助函数,来为我们完成一些最常见的操作。采用这种方法,我们不必一再重复相同的代码,也不必一再犯同样的错误,以及查找、修正这些错误。几乎对于每个类型,初始化和赋值都属于最常用的操作。对 Date 来说,增加日期的值是另一个常用操作,于是我们可以编写如下辅助函数:

```
// helper functions:

void init_day(Date& dd, int y, int m, int d)
{
    // check that (y,m,d) is a valid date
    // if it is, use it to initialize dd
}

void add_day(Date& dd, int n)
{
    // increase dd by n days
}
```

现在我们可以试着使用 Date 了:

```

void f()
{
    Date today;
    init_day(today, 12, 24, 2005);    // oops! (no day 2005 in year 12)
    add_day(today, 1);
}

```

首先,我们注意到这些操作(这里实现为辅助函数)是很有用的。如果我们没有一劳永逸地编写一个日期检查程序的话,检查日期将会是非常困难和乏味的,我们有时可能会忘记写检查代码,从而得到充斥错误的程序。每当定义一个类型时,我们都会需要一些针对该类型对象的操作。而到底需要多少个操作,需要什么类型的操作,不同的类型各不相同。我们如何实现这些操作(实现为函数或成员函数或运算符)也是不同的。但只要是决定定义一个类型,我们都要问一下自己:“我们想要为这个类型设计什么样的操作?”

9.4.2 成员函数和构造函数

我们为 Date 设计了一个初始化函数,它提供了重要的日期合法性检查功能。然而,如果我们使用不当的话,日期检查功能将毫无用处。例如,假定我们已经为 Date 定义了输出运算符 << (参见 9.8 节):

```

void f()
{
    Date today;
    // ...
    cout << today << '\n';    // use today
    // ...
    init_day(today, 2008, 3, 30);
    // ...
    Date tomorrow;
    tomorrow.y = today.y;
    tomorrow.m = today.m;
    tomorrow.d = today.d + 1;    // add 1 to today
    cout << tomorrow << '\n';    // use tomorrow
}

```

这段代码中,我们定义 today 后,“忘记了”立即对它进行初始化,而“某人”在我们及时调用 init_day() 之前就使用了它。而且“某人”认为调用 add_day() 浪费时间,或许他根本没听说过这个函数,因此他亲手构造了 tomorrow 而不是调用 add_day()。由于这些情况碰巧发生,这个程序变成了一段问题代码——而且问题非常严重。有时,而且可能是大多数时候,它工作正常,但一些小的改变就可能导致严重的错误。例如,一个未初始化的 Date 会产生垃圾输出,而简单地为成员变量 d 加 1 来推移日期会成为定时炸弹:当 today 表示月底那一天时,加 1 操作会导致一个非法的日期。这段“问题严重的代码”最大的问题是,它看起来似乎没什么问题。

上述思考促使我们寻找更好的操作实现方式,我们需要不会被程序员忘记的初始化函数,需要被忽视的可能性很低的操作。实现这些目标的基本技术就是成员函数(member function),即将函数声明于类体,作为类的成员。例如:

```

// simple Date
// guarantee initialization with constructor
// provide some notational convenience
struct Date {
    int y, m, d;    // year, month, day
    Date(int y, int m, int d);    // check for valid date and initialize
    void add_day(int n);    // increase the Date by n days
};

```

与类同名的成员函数是特殊的成员函数，称为构造函数 (constructor)，专门用于类对象的初始化 (“构造”)。如果一个类具有需要参数的构造函数，而程序员忘记利用它初始化类对象，则编译器会捕获这个错误。C++ 提供了一种专用的，而且很方便的语法来进行这种初始化，例如：

```
Date my_birthday;           // error: my_birthday not initialized
Date today(12,24,2007);      // oops! run-time error
Date last(2000, 12, 31);     // OK (colloquial style)
Date christmas = Date(1976,12,24); // also OK (verbose style)
```

试图声明 `my_birthday` 的语句是错误的，因为我们没有指定所需的初值。试图声明 `today` 的语句会编译通过，但构造函数中的检查代码会在运行时捕获非法的日期 (12 年 24 月 2007 日，不存在这样的日期)。

定义 `last` 的语句提供了初值——`Date` 的构造函数所需的参数，位置是在紧跟变量名的括号中。对于一个具有带参数的构造函数的类，这是最常见的类变量初始化方式。我们也可以用一种更为啰嗦的方式：显式地创建一个对象 (在此处，是 `Date(1976, 12, 24)`)，然后通过赋值方式用此初值对变量进行初始化，如上面代码中对 `christmas` 的初始化。除非你确实喜欢打字，否则你很快就会厌烦这种方式。

现在，我们可以试着使用新定义的这些变量：

```
last.add_day(1);
add_day(2);           // error: what date?
```

注意，成员函数 `add_day()` 必须对特定的 `Date` 对象进行调用，语法是使用成员访问符号“.”。我们会在 9.4.4 节介绍如何定义一个成员函数。

9.4.3 保持细节私有性

现在，我们还有一个问题没有解决：如果有人忘了使用成员函数 `add_day()` 怎么办？如果有人决定直接修改月份怎么办？毕竟，我们“忘了”提供这些功能：

```
Date birthday(1960,12,31); // December 31, 1960
++birthday.d;           // ouch! invalid date

Date today(1970,2,3);
today.m = 14;           // ouch! invalid date
```

只要我们还是将 `Date` 的描述暴露给所有人，那么就会有人 (无意或有意地) 把事情搞乱——也就是制造出非法的日期值。例如上面的代码，就创建了日历上不存在的日期。这样的非法对象会成为定时炸弹：在有人使用它之前，它只不过是时间值，但一旦有人使用这样的非法时间，就会发生运行时错误，而通常情况会更糟，程序会生成错误的结果。

上述担忧使我们得到如下结论：`Date` 的描述对用户来说应该是不可访问的，除非是通过类中提供的公有成员函数来访问。下面是按这种思想改进后的第一个版本：

```
// simple Date (control access)
class Date {
    int y, m, d;           // year, month, day
public:
    Date(int y, int m, int d); // check for valid date and initialize
    void add_day(int n);       // increase the Date by n days
    int month() { return m; }
    int day() { return d; }
    int year() { return y; }
};
```

使用新版 `Date` 的示例如下：

```
Date birthday(1970, 12, 30);           // OK
birthday.m = 14;                       // error: Date::m is private
cout << birthday.month() << endl;      // we provided a way to read m
```

“有效日期”的概念是有效值思想的一个重要特例。我们在设计类型时，设法保证有效值。即我们隐藏类描述，提供一个创建有效对象的构造函数，所有成员函数的设计也遵循接受有效值、生成有效值的原则。对象的值通常称为状态，因此，有效值的思想通常称为对象的有效状态。

我们可以不在每次使用对象时都进行有效性检查，代之以期望没有人到处散布无效值。经验表明，“期望”可以导致“很漂亮的”程序。但是，“很漂亮的”程序偶尔会产生错误的结果或者崩溃，因此，作为一名专业人员，编写这样的程序是不能赢得朋友和声望的。我们宁愿编写那种可被证明正确性的代码。

判定有效值的规则称为不变式(invariant)。Date 的不变式——“一个 Date 对象必须表示过去、现在或将来的某一天”是一个少见的例子，它很难表述准确：我们需要考虑闰年、格里高利历法、时区等。但是，如果只是用于特定实际应用，我们还是可以给出 Date 的不变式的。例如，如果分析互联网日志，我们就无须为格里高利历、儒略历或是玛雅历而困扰。如果我们不能想出一个完美的不变式，那我们可能就要处理普通数据。如果是这样，我们可以使用 struct。

9.4.4 定义成员函数

到目前为止，我们已经以一个接口设计者和一个用户的角度考察了 Date，但我们迟早要实现这些成员函数。第一步，我们先给出 Date 类声明的一个重新组织过的子集，它展示了适合于描述公共接口的一般风格：

```
// simple Date (some people prefer implementation details last)
class Date {
public:
    Date(int y, int m, int d); // constructor: check for valid date and initialize
    void add_day(int n);       // increase the Date by n days
    int month();
    // ...
private:
    int y, m, d;              // year, month, day
};
```

人们把公共接口放在类的开始，是因为接口是大多数人最感兴趣的。理论上，用户无需了解类的实现细节，只需知道接口即可。实际上，我们通常会有好奇心，会快速浏览一下类的实现，看看它是否合理，我们是否能从中学到一些技术。但是，除非我们就是实现者，否则我们会倾向于在公有接口上花更多的时间。编译器并不关心类成员的顺序，你想以什么样的顺序来声明它们，编译器都能接受。

```
Date::Date(int yy, int mm, int dd) // constructor
    :y(yy), m(mm), d(dd)           // note: member initializers
{
}

void Date::add_day(int n)
{
    // ...
}

int month() // oops: we forgot Date::
{
    return m; // not the member function, can't access m
}
```

符号: `y(yy)`, `m(mm)`, `d(dd)` 就是类成员初始化的语法。当然也可以这样写:

```
Date::Date(int yy, int mm, int dd)    // constructor
{
    y = yy;
    m = mm;
    d = dd;
}
```

但后一种写法,原则上讲,是先用默认值对成员进行了初始化,然后又对它们进行了赋值。而且这种写法的一个潜在问题是,我们有可能无意地在成员初始化之前使用它们。`y(yy)`, `m(mm)`, `d(dd)` 这种方式更直接地表达了我们的意图。两种写法之间的区别与下面两段代码之间的区别是一样的:

```
int x;    // first define the variable x
// ...
x = 2;    // later assign to x
```

和

```
int x = 2;    // define and immediately initialize with 2
```

出于一致性的考虑,甚至第二段代码中的初始化语句也可以用“参数”/括号的语法来表达:

```
int x(2);    // initialize x with 2
Date sunday(2004,8,29);    // initialize sunday with (2004,8,29)
```

我们也可以直接在类定义中定义成员函数:

```
// simple Date (some people prefer implementation details last)
class Date {
public:
    Date(int yy, int mm, int dd)
        :y(yy), m(mm), d(dd)
    {
        // ...
    }

    void add_day(int n)
    {
        // ...
    }

    int month() { return m; }

    // ...
private:
    int y, m, d;    // year, month, day
};
```

我们需要注意的第一点是,将成员函数定义放在类定义中会使类声明变得大而“凌乱”。例如在此例中,构造函数和 `add_day()` 的代码会有十几行甚至更长。这使类声明的规模比原来增大几倍,而且使用户难以在实现细节中找到接口。因此,我们不会在类声明中定义大的函数。

但是,看一下 `month()` 的定义,它比放在类声明之外的版本 `Date::month()` 要更为直接和简短。对于这种简单的、较小的函数,我们应该考虑直接在类声明中给出其定义。

注意, `month()` 可以访问定义在其后(下)面的 `m`。实际上,类成员对其他成员的访问并不依赖于成员在类中的声明位置。本书前文介绍的“名字必须在使用之前声明”这一规则,在一个类内的有限作用域中可以放宽。

将成员函数的定义放在类定义内有两个作用:

- 函数将成为内联的 (inlined)，即编译器为此函数的调用生成代码时，不会生成真正的函数调用，而是将其代码嵌入到调用者的代码中。对于 month() 这种所做工作很少，又被频繁调用的函数，这种编译方式会带来很大的性能提升。
- 每当我们对内联函数体作出修改时，所有使用这个类的程序都不得不重新编译。如果函数体位于类声明之外的话，就不必这样，只在类接口改变时才需要重新编译用户程序。对于大程序来说，函数体改变时无需重新编译程序会是一个巨大的优势。

显然，我们应遵循如下基本原则：除非你明确需要从小函数的内联中获得性能提升，否则不要将成员函数体放在类声明中。对于 5 行以上代码的函数，不会从内联中获益。对于由超过一两个表达式组成的函数，本书很少采用内联方式。

9.4.5 引用当前对象

考虑如下使用 Date 类的简单代码：

```
class Date {
    // ...
    int month() { return m; }
    // ...
private:
    int y, m, d; // year, month, day
};

void f(Date d1, Date d2)
{
    cout << d1.month() << ' ' << d2.month() << '\n';
}
```

Date::month() 是如何知道第一次被调用时应打印 d1.m，第二次被调用时应打印 d2.m 呢？再看一下 Date::month()，其声明指出它没有任何参数！那么 Date::month() 是怎么知道是哪个对象在调用它呢？奥妙在这里：每个类成员函数（如 Date::month()）都有一个隐式参数，用来识别调用它的对象。因此，在第一次调用中，m 会正确地指向 d1.m，而在第二次调用中，它指向 d2.m。参见 17.10 节，获得更多使用此隐式参数的内容。

9.4.6 报告错误

当我们发现一个无效日期时，应该做什么呢？检查无效日期的代码应该放在程序中什么位置呢？从 5.6 节我们可以得到第一个问题的答案：“抛出一个异常”，而放置检查代码的位置显然应该是我们最初构造一个 Date 对象时。如果我们没有创建无效的 Date 对象，而且成员函数也编写正确，那么我们就永远不会得到具有无效值的 Date 对象。因此，我们应该阻止用户创建具有无效状态的 Date 对象：

```
// simple Date (prevent invalid dates)
class Date {
public:
    class Invalid { }; // to be used as exception
    Date(int y, int m, int d); // check for valid date and initialize
    // ...
private:
    int y, m, d; // year, month, day
    bool check(); // return true if date is valid
};
```

我们将有效性检查代码放到一个独立的函数 check() 中，这一方面是因为，从逻辑上讲，有效性检查与初始化就是不同的工作，另一方面是因为，我们可能需要多个构造函数。如你所见，除了私

有数据外，我们还可以为类声明私有函数：

```
Date::Date(int yy, int mm, int dd)
    : y(yy), m(mm), d(dd)           // initialize data members
{
    if (!check()) throw Invalid();   // check for validity
}

bool Date::check() // return true if date is valid
{
    if (m<1 || 12<m) return false;
    // ...
}
```

给出这样的 Date 定义后，我们可以写出如下代码：

```
void f(int x, int y)
try {
    Date dxy(2004,x,y);
    cout << dxy << '\n';           // see §9.8 for a declaration of <<
    dxy.add_day(2);
}
catch(Date::Invalid) {
    error("invalid date");           // error() defined in §5.6.3
}
```

我们现在知道，<< 和 add_day() 会获得一个有效的日期作为它们的操作对象。

我们将在 9.7 节完成 Date 类的演化，在此之前，我们先介绍几个常用的语言功能，我们需要这些功能来更好地完成 Date 类的演化：枚举类型和运算符重载。

9.5 枚举类型

枚举 (enumeration, 简称为 enum) 是一种非常简单的用户自定义类型，它指定一个值的集合，这些值用符号常量表示，称为枚举量 (enumerator)。下面是一个例子：

```
enum Month {
    jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};
```

一个枚举定义的“体”就是一个简单的枚举量列表。你可以为枚举量指定特定的值，就像上面代码为 jan 指定值一样。也可以不指定，让编译器选择合适的值。如果你让编译器来选择值，它赋予每个枚举量的值为上一个枚举量的值加上 1。因此，上面 Month 的定义赋予月份从 1 开始的连续整数。此定义与下面的定义是等价的：

```
enum Month {
    jan=1, feb=2, mar=3, apr=4, may=5, jun=6,
    jul=7, aug=8, sep=9, oct=10, nov=11, dec=12
};
```

但是，第二种定义一方面冗长乏味，另一方面容易出错。实际上，我们在输入第二个定义时出现了两个错误，经过修改后才得到上面的正确版本。因此，最好还是让编译器来做这种简单的、重复性的、“机械的”工作。编译器比我们更擅长这种工作，而且它不会厌烦。

如果我们不初始化第一个枚举量，那么编译器会从 0 开始计数。例如：

```
enum Day {
    monday, tuesday, wednesday, thursday, friday, saturday, sunday
};
```

这里 monday == 0，而 sunday == 6。在实践中，从 0 开始往往是一种更好的选择。

我们可以像下面代码那样使用 Month：

```

Month m = feb;
m = 7;           // error: can't assign an int to a Month
int n = m;       // OK: we can get the numeric value of a Month
Month mm = Month(7); // convert int to Month (unchecked)

```

注意, Month 是一个独立的类型, 它可以隐式转换为整型, 但整型不能隐式转换为 Month 类型。这是合理的, 因为每个 Month 值都对应一个相等的整型值, 但很多整型值是没有相等的 Month 值的。

例如, 我们肯定希望下面的初始化失败:

```
Month bad = 9999; // error: can't convert an int to a Month
```

如果你非要坚持使用 Month(9999) 这种语法, 那么一切结果得由你自己负责! 很多情况下, 如果程序员明确地坚持做一些可能很蠢的事情, C++ 不会设法阻止, 毕竟程序员可能更清楚自己在做什么。

遗憾的是, 我们不能为枚举类型定义一个构造函数来检查初始值, 但编写一个简单的检查函数还是很容易的:

```

Month int_to_month(int x)
{
    if (x < jan || dec < x) error("bad month");
    return Month(x);
}

```

有了这个函数, 我们就可以这样来检查初始值了:

```

void f(int m)
{
    Month mm = int_to_month(m);
    // ...
}

```

我们能枚举类型做什么呢? 原则上, 枚举类型可以用于任何需要一组相关的命名整型常量的地方。当我们想表示选项(如上、下; 是、否、也许; 开、关; 北、东北、东、东南、南、西南、西、西北等)或表示不同值(如红、蓝、绿、黄、栗、深红、黑)时, 就属于这种情况。

值得注意的是, 枚举量的作用域不局限在其枚举类型内, 而是与其枚举类型有着相同的作用域。例如:

```

enum Traffic_sign { red, yellow, green };
int var = red; // note: not Traffic_sign::red

```

这可能会带来问题。如果你为全局名字起了一些很短而且常用的名字, 如 red、on、ne 和 dec, 就有可能造成混淆。例如, ne 的意思是“northeast”(东北)还是“not equal”(不等)? dec 的意思是“decimal”(十进制数)还是“December”(十二月)? 这是我们在 3.7 节就告诫过大家的一类问题, 如果我们定义枚举类型时使用了一些短的、常用的名字, 而其作用域又是全局作用域的话, 就很容易遇到这种问题。实际上, 当我们试图同时使用 Month 和 istream 时, 就会立刻遇到这种名字冲突, 因为 istream 中有一个名为 dec 的“操作符”, 表示“decimal”(参见 11.2.1 节)。为了避免这种问题, 我们通常倾向于将枚举类型定义于更窄的作用域中, 如一个类中。除了避免名字冲突外, 这还使枚举值表示的含义更为明确, 如 Month::jan 和 Color::red。我们将在 9.7.1 节中介绍这种计数。如果确实需要全局名字, 我们会通过增加名字长度、使用不常用的名字(或不常用的拼写方式)以及使用大小写等方式来设法降低名字冲突的机会。然而, 我们的首选方案还是在合理的前提下尽量将名字放在更局部的作用域中。

9.6 运算符重载

你可以在类或枚举对象上定义几乎所有 C++ 运算符, 这通常称为运算符重载(operator over-

loading)。这种机制用于为用户自定义类型提供习惯的符号表示方法。例如：

```
enum Month {
    Jan=1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec
};

Month operator++(Month& m)           // prefix increment operator
{
    m = (m==Dec) ? Jan : Month(m+1); // "wrap around"
    return m;
}
```

其中?:是“算术 if”运算符：当(m==Dec)时 m 的值变为 Jan，否则 m 的值为 Month(m+1)。这是十二月后“绕回”一月这一事实的一种非常简洁的描述方法。我们可以像如下代码一样使用 Month 类型：

```
Month m = Sep;
++m;      // m becomes Oct
++m;      // m becomes Nov
++m;      // m becomes Dec
++m;      // m becomes Jan ("wrap around")
```

你可能觉得增加 Month 对象值这样的操作没那么常用，不至于设计一个专门的运算符。可能确实是这样，那么输出运算符又如何呢？如下代码定义了一个输出运算符：

```
vector<string> month_tbl;

ostream& operator<<(ostream& os, Month m)
{
    return os << month_tbl[m];
}
```

这里假定 month_tbl 已经在其他位置进行了初始化，每个数组元素保存了对应月份的恰当的名字，如 month_[Mar] 的值为字符串“March”，参见 10.11.3 节。

你可以为自己的类型重新定义几乎所有的 C++ 运算符，如 +、-、*、/、%、[]、()、^、!、&、<、<=、> 和 >= 等。你不能定义新的运算符，你可能想在程序中把 * 或 \$ = 作为运算符，但 C++ 不允许你这样做。而且，重载运算符时，运算对象数目也必须与原来一样。例如，你可以定义一元运算符 -，但不能定义一元的 <= (小于等于)，你可以定义二元运算符 +，但不能定义二元的 ! (非)。原则上，C++ 允许你对自定义类型使用已有的语法，但不允许扩展语法。

一个重载的运算符必须作用于至少一个用户自定义类型的运算对象：

```
int operator+(int,int); // error: you can't overload built-in +
Vector operator+(const Vector&, const Vector &); // OK
Vector operator+=(const Vector&, int); // OK
```

一个一般性的原则是：除非你真正确定重载运算符能大大改善代码，否则不要为你的类型定义运算符。而且，重载运算符应该保持其原有意义：+ 就应该是加法，二元运算符 * 就应该表示乘法，[] 表示元素访问，() 表示调用，等等。这只是建议，并不是 C++ 语言规则，但这是一个有益的建议：按习惯使用运算符，例如 + 只用做加法，对我们理解程序会有极大的帮助。毕竟，这种习惯用法源于人们千百年来使用数学符号的经验。相反，含混不清的运算符和与常规不符的使用方式是混乱和错误之源。

注意，就像大家一般猜想的那样，重载最多的运算符是 +、-、*、/，而不是 =、==、!=、<、[] 及 ()。

9.7 类接口

我们已经讨论过,类的公有接口和实现部分应该分离。只要为那些“旧式简单数据”类型保留着 struct 功能,不同意接口和实现分离原则的专业人员就会很少。但是,如何来设计一个好的接口呢?好的公共接口和乱糟糟的接口之间有什么区别呢?回答这个问题必须借助实例,但我们仍能列出一些一般原则,它们在 C++ 中都有支持:

- 保持接口的完整性。
- 保持接口的最小化。
- 提供构造函数。
- 支持(或者禁止)拷贝(参见 14.2.4 节)。
- 使用类型来提供完善的类型检查。
- 支持不可修改的成员函数(参见 9.7.4 节)。
- 在析构函数中释放所有资源(参见 17.5 节)。

参见 5.5 节(如何检测及报告运行时错误)。

前两条原则可以归结为“保持接口尽可能小,但不要更小了”。我们希望接口尽量小,是因为小的接口易于学习和记忆,而实现者也不会为不必要的和很少使用的功能浪费大量时间。小的接口还意味着当错误发生时,我们只需检查很少的函数来定位错误。平均来看,公有成员函数越多,查找 bug 就越困难——调试带有公有数据的类是非常复杂的,不要让陷入其中。当然,前提还是要保证完整性,否则接口就没有用处了。如果一个接口无法完成我们真正需要做的全部工作,我们是不会使用它的。

下面我们讨论其他一些话题,这些话题更为具体,直接对应 C++ 语言功能。

9.7.1 参数类型

当我们在 9.4.3 节中为 Date 定义构造函数时,使用了三个整型作为其参数。这会带来一些问题:

```
Date d1(4,5,2005);    // oops: year 4, day 2005
Date d2(2005,4,5);    // April 5 or May 4?
```

第一个问题(无效的日和月)比较容易处理,在构造函数中进行检测即可。但是,第二个问题(月和日的混淆),通过用户编写的检测代码是无法查找出来的。这个问题是由于人们书写月和日的习惯不同而造成的:例如,4/5 在美国表示 4 月 5 日,但在英国表示 5 月 4 日。我们不能指望不遇到这个问题,必须采取其他手段解决它。一种明显的解决方案是使用类型系统:

```
// simple Date (use Month type)
class Date {
public:
    enum Month {
        jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
    };

    Date(int y, Month m, int d);    // check for valid date and initialize
    // ...

private:
    int y;        // year
    Month m;
    int d;        // day
};
```

如果像上面代码一样使用了 Month 类型, 当我们颠倒了月和日这两个参数时, 编译器就会捕获这个问题。而且, 使用一个枚举类型作为月的类型, 令我们可以使用符号名来表示月份, 这样的代码通常比直接使用数字更易读, 因而也更容易出错:

```
Date dx1(1998, 4, 3);           // error: 2nd argument not a Month
Date dx2(1998, 4, Date::mar);   // error: 2nd argument not a Month
Date dx2(4, Date::mar, 1998);   // oops: run-time error: day 1998
Date dx2(Date::mar, 4, 1998);   // error: 2nd argument not a Month
Date dx3(1998, Date::mar, 30);   // OK
```

在这段代码中, 编译器帮我们避免了很多“意外”。注意代码中用类名 Date 来限定枚举量 mar 的部分: Date::mar。这就是我们所说的, mar 是 Date 的 mar。我们没有用 Date.mar, 因为 Date 不是一个对象(而是一个类型), 而 mar 也不是一个数据成员(而是一个枚举量——一个符号常量)。我们在类名(或名字空间名, 参见 8.7 节)后使用::, 而在对象名后使用.(点)。

如果可以选择, 我们宁可在编译时将错误捕获, 而不要留在运行时。我们更希望由编译器找到错误, 而不是由我们自己来寻找到底是哪部分代码发生了问题。而且, 如果错误可以在编译时被捕获, 我们就不需要编写检查代码, 最终程序的效率也就会更高。

考虑这么一个问题: 我们能捕获颠倒日/月与年的情况吗? 当然是可以的, 但是解决方案不像处理日与月的颠倒那么简单、优美。毕竟, 像公元 4 年这样的年份也是有效的, 我们的方案必须能表示这样的年份。即使将日期限定为近代, 需要表示的年份也实在是太多了, 无法定义一个枚举来描述所有这些年份。

可能我们能做到的最好程度(在不了解很多 Date 用途的情况下), 像下面代码这样:

```
class Year {           // year in [min:max] range
    static const int min = 1800;
    static const int max = 2200;
public:
    class Invalid { };
    Year(int x) : y(x) { if (x<min || max<x) throw Invalid(); }
    int year() { return y; }
private:
    int y;
};

class Date {
public:
    enum Month {
        jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
    };

    Date(Year y, Month m, int d);   // check for valid date and initialize
    // ...
private:
    Year y;
    Month m;
    int d;   // day
};
```

现在, 我们可以在编译时避免年份颠倒的错误了:

```
Date dx1(Year(1998), 4, 3);           // error: 2nd argument not a Month
Date dx2(Year(1998), 4, Date::mar);   // error: 2nd argument not a Month
Date dx2(4, Date::mar, Year(1998));   // error: 1st argument not a Year
Date dx2(Date::mar, 4, Year(1998));   // error: 2nd argument not a Month
Date dx3(Year(1998), Date::mar, 30);   // OK
```

当然，下面这种怪异的，不太可能出现的错误，编译时仍无法捕获：

```
Date dx2(Year(4), Date::mar, 1998); // run-time error: Year::Invalid
```

做这些额外的工作、引入这些新的符号来进行年份的检查是否值得呢？这自然取决于你用 `Date` 解决什么问题。但在本书中，我们认为没有必要这么做，因此后面不会使用 `Year` 类。

当我们编程时，应该一直问自己：对于给定的应用，什么是足够好的解决方案？我们通常不会奢侈到，在已经得到一个足够好的方案后，还会“无止境地”去追求最佳方案。继续追寻下去的话，我们甚至可能得到一些非常复杂，但却比最初的简单方案更差的方案。人们常说的“至善者善之敌”就是这个意思。

注意 `min` 和 `max` 定义中对 `static const` 的使用。这就是我们在类中定义整型符号常量的方法。我们使用 `static` 限定一个类成员，可以保证它在整个程序中只有一份拷贝，而不是每个类对象都有一份。

9.7.2 拷贝

编程中总是要创建对象的，也就是说，我们总是要考虑初始化和构造函数。构造函数可能是最重要的类成员：为了编写构造函数，我们必须确定初始化一个对象时应该做什么，以及什么样的值是有效值（什么是不变式）？单纯地考虑初始化工作，会帮助你在设计构造函数时避免错误。

下一个经常要考虑的问题是：我们需要拷贝对象吗？如果需要，如何来拷贝呢？

对于 `Date` 或 `Month`，答案是我们显然需要拷贝这两种类型的对象；而这两种类型的对象的拷贝的含义很简单——只要复制所有成员即可。实际上，这正是默认情况。只要你不特别声明，编译器就会正确地做到上述效果。例如，如果你将一个 `Month` 对象作为初始化值和赋值运算的右部，编译器就会完成其所有成员的拷贝：

```
Date holiday(1978, Date::jul, 4); // initialization
Date d2 = holiday;
Date d3 = Date(1978, Date::jul, 4);
holiday = Date(1978, Date::dec, 24); // assignment
d3 = holiday;
```

这段代码已经完全按我们的期望工作了。用 `Date(1978, Date::dec, 24)` 可以创建一个正确的未命名 `Date` 对象，你可以用它来做一些适当的工作。例如：

```
cout << Date(1978, Date::dec, 24);
```

这里对构造函数的使用，从字面上看，与一个类的作用非常相近。通常，当我们需要定义一个只使用一次的变量或常量时，这是一种很方便的方法。

如果我们需要拷贝操作的含义与默认情况不同，应该怎么做呢？我可以定义自己的拷贝函数（参见 18.2 节），或者把拷贝构造函数和拷贝赋值运算符设置为私有的（参见 14.2.4 节）。

9.7.3 默认构造函数

未初始化的变量可能会成为错误之源。为了解决这个问题，我们可以用构造函数来保证类的每个对象都被初始化。例如，我们定义了构造函数 `Date::Date(int, Month, int)` 来保证每个 `Date` 对象都会被正确地初始化。这意味着程序员必须提供三个类型正确的参数。例如：

```
Date d1; // error: no initializer
Date d2(1998); // error: too few arguments
Date d3(1,2,3,4); // error: too many arguments
Date d4(1,"jan",2); // error: wrong argument type
Date d5(1,Date::jan,2); // OK: use the three-argument constructor
Date d6 = d5; // OK: use the copy constructor
```

注意，虽然我们为 `Date` 定义了一个需要三个参数的构造函数，但是通过赋值运算直接拷贝还是没

有问题的。

很多类都能很好地理解默认值，即它们能解决这个问题：“如果我没有为对象提供一个初始值，那么它应该具有什么值？”例如：

```
string s1;           // default value: the empty string ""
vector<string> v1;    // default value: the empty vector; no elements
vector<string> v2(10); // vector of 10 default strings
```

这些代码就是像注释所描述的那样工作，这看起来很合理。之所以 `vector` 和 `string` 类能支持这样的特性，是因为它们都有默认构造函数，可以隐含地进行所需的初始化工作。

对于类型 `T`，符号 `T()` 表示默认值，这是通过定义默认构造函数实现的，因此，我们可以写出下面这样的代码：

```
string s1 = string();           // default value: the empty string ""
vector<string> v1 = vector<string>(); // default value:
                                   // the empty vector; no elements
vector<string> v2(10, string()); // vector of 10 default strings
```

但是，我们更倾向于采用下面这种等价的和更“口语化”的语法形式：

```
string s1;           // default value: the empty string ""
vector<string> v1;    // default value: the empty vector; no elements
vector<string> v2(10); // vector of 10 default strings
```

对于内置类型，如 `int` 和 `double` 来说，默认构造函数的结果为 0，即 `int()` 是 0 的一种复杂描述，而 `double()` 是 0.0 的一种啰嗦的说法。

请小心——用 `()` 语法进行初始化存在一个令人讨厌的问题，例如：

```
string s1("lke"); // string initialized to "lke"
string s2();      // function taking no argument returning a string
```

使用默认构造函数不只是形式上的问题，它有着更深层次的重要作用。设想一下，我们可能会得到未初始化的 `string` 或 `vector` 对象：

```
string s;
for (int i=0; i<s.size(); ++i) // oops: loop an undefined number of times
    toupper(s[i]); // oops: modify the contents of a random memory location

vector<string> v;
v.push_back("bad"); // oops: write to random address
```

如果 `s` 和 `v` 的值真的是未定义，我们就完全不知道它们包含多少个元素或者无法（采用一些常用的实现技术，参见 17.5 节）知道这些元素存放在哪里。其结果就是我们可能会访问随机地址——这会导致最麻烦的一类错误。基本上，没有构造函数，我们就无法建立一个不变式，也就不能保证变量中的值是有效的（参见 9.4.3 节）。我们必须坚持对变量进行初始化，必须坚持使用初始化程序，像下面代码这样：

```
string s1 = "";
vector<string> v1(0);
vector<string> v2(10, ""); // vector of 10 empty strings
```

但是，这种方法也不是非常完美。对于 `string` 类型，非常显然 `""` 是表示“空字符串”。对于 `vector` 类型，用 0 来表示“空向量”也不会引起什么混淆。然而，对于很多类型来说，找到一个值，能合理地表示默认值并不那么容易。因此，更好的方法是定义一个构造函数，不需要程序员显式提供初始化程序就能创建对象。这种构造函数不接受参数，我们称之为默认构造函数。

例如，对于日期来说，就不存在一个显然的默认值。这就是我们为什么到目前为止还没有为 `Date` 定义一个默认构造函数的原因，现在我们为它定义一个（只是说明我们可以这样做而已）：


```

class Date {
public:
    // ...
    Date(); // default constructor
    // ...
private:
    int y;
    Month m;
    int d;
};

```

我们需要挑选一个默认日期，21 世纪的第一天可能是个合理的选择：

```

Date::Date()
    :y(2001), m(Date::jan), d(1)
{
}

```

如果我们不想把默认值写入构造函数的代码中，我们可以使用一个常量（或一个变量）。为了避免使用全局变量带来的初始化等问题，我们使用 8.6.2 节中介绍的技术：

```

Date& default_date()
{
    static Date dd(2001,Date::jan,1);
    return dd;
}

```

这里使用了 static，这样就不会每次调用 default_date() 时都创建变量 dd，它只在第一次调用 default_date() 时被创建并被初始化。有了 default_date() 函数，为 Date 创建一个默认构造函数就很简单了：

```

Date::Date()
    :y(default_date().year()),
    m(default_date().month()),
    d(default_date().day())
{
}

```

注意，默认构造函数无须检查对象值，因为创建默认日期的函数已经做过了。有了 Date 的构造函数，我们就可以定义 Date 数组了：

```
vector<Date> birthdays(10);
```

如果没有默认构造函数，我们可能不得不这样做：

```
vector<Date> birthdays(10,default_date());
```

9.7.4 const 成员函数

对于有些变量，我们希望它们可以被改变——这也是为什么我们称之为“变量”的原因。但对于另外一些变量，我们则不希望改变它们，即我们想用“变量”表示的实际上是不变量。我们通常称之为常量（constant，或者简写为 const）。考虑下面代码：

```

void some_function(Date& d, const Date& start_of_term)
{
    int a = d.day();           // OK
    int b = start_of_term.day(); // should be OK (why?)
    d.add_day(3);              // fine
    start_of_term.add_day(3);   // error
}

```

在这里，我们希望 d 是可变的，start_of_term 是不可变的，而 some_function() 将不被允许对 start_of_term 进行更改。编译器是如何知道这些的呢？这是因为我们将 start_of_term 定义为常量的，从而使编译器获得了上述信息。好了，我们达到了预期的目的，但是，为什么用 day() 来读取 start_

of_term 的成员 day 是被允许的呢？根据前面给出的 Date 的定义，因为编译器不知道 day() 是否修改了对象的日期，start_of_term.day() 应该是错误的。我们没有给出过这方面的任何信息，因此编译器应该假定 day() 有可能改变日期，它应该报告一个错误。

我们可以将类操作划分为两类：可更改和不可更改，这样就可以解决这个问题了。这个语言特性对于我们深入理解类是非常重要的，而且它也具有很重要的实践意义：不修改对象的操作可以在常量对象上调用，例如：

```
class Date {
public:
    // ...
    int day() const;           // const member: can't modify the object
    Month month() const;       // const member: can't modify the object
    int year() const;          // const member: can't modify the object

    void add_day(int n);        // non-const member: can modify the object
    void add_month(int n);      // non-const member: can modify the object
    void add_year(int n);       // non-const member: can modify the object
private:
    int y;                     // year
    Month m;
    int d;                     // day of month
};

Date d(2000, Date::jan, 20);
const Date cd(2001, Date::feb, 21);

cout << d.day() << " - " << cd.day() << endl; // OK
d.add_day(1); // OK
cd.add_day(1); // error: cd is a const
```

在一个成员函数声明中，我们将 const 放置参数列表右边，就表示这个成员函数可以在一个常量对象上调用。一旦我们将一个成员函数声明为常量的，编译器会帮助我们保证这个成员函数不会修改对象。例如：

```
int Date::day() const
{
    ++d; // error: attempt to change object from const member function
    return d;
}
```

当然，我们不会经常像上面代码这样“欺骗”编译器。但我们可能会无意中这么做，特别是当代码非常复杂时，而编译器可以保证避免这样的问题。

9.7.5 类成员和“辅助函数”

当我们试图最小化类接口时（在保证完整性的前提下），不得不忽略大量有用的操作。如果一个函数可以简单、优美、高效地实现为一个独立函数时（即实现为非成员函数），就应该将它的实现放在类外。采用这种方式，函数中的错误就不会直接破坏类对象中的数据。不访问类描述是很重要的，因为常用的 debug 技术是“首先排查惯犯”，即当类出现问题时，首先检查直接访问类描述的函数：几乎可以肯定是这类函数导致的错误。如果这类函数只有十几个而不是 50 个的话，我们当然会很高兴。

Date 类有 50 个成员函数！你一定认为我们是在开玩笑。但我们没有：几年前我调查了一些商用的 Date 库，发现这些库中充斥着像 next_Sunday()、next_workday() 等这样的函数。对于一个设计目标更倾向于方便用户使用，而不是易于理解、实现和维护的类来说，有 50 个成员函数并不过分。

另一点值得注意的是，如果类描述改变，只有直接访问描述的函数才需要重写。这是保持接口最小化的另一个重要原因。在我们的 `Date` 例子中，我们可能会觉得用一个整数表示自 1900 年 1 月 1 日至今的天数，比(年, 月, 日)的形式好得多。如果做出这样的改变，只需要修改成员函数。

下面是一些辅助函数(helper function)的例子：

```
Date next_Sunday(const Date& d)
{
    // access d using d.day(), d.month(), and d.year()
    // make new Date to return
}

Date next_weekday(const Date& d) { /* ... */ }

bool leapyear(int y) { /* ... */ }

bool operator==(const Date& a, const Date& b)
{
    return a.year()==b.year()
           && a.month()==b.month()
           && a.day()==b.day();
}

bool operator!=(const Date& a, const Date& b)
{
    return !(a==b);
}
```

辅助函数也有便利函数(convenience function)、帮助函数(auxiliary function)等很多名字。这类函数和其他非成员函数在逻辑上是有区别的——辅助函数是一种设计思想，而不是一种编程概念。辅助函数通常接受一个类对象作为其参数，它就是为这个类做辅助工作。当然也有例外：注意 `leapyear()`。我们通常用命名空间来区分一组辅助函数，参见 8.7 节：

```
namespace Chrono {
    class Date { /* ... */ };
    bool is_date(int y, Date::Month m, int d); // true for valid date
    Date next_Sunday(const Date& d) { /* ... */ }
    Date next_weekday(const Date& d) { /* ... */ }
    bool leapyear(int y) { /* ... */ } // see exercise 10
    bool operator==(const Date& a, const Date& b) { /* ... */ }
    // ...
}
```

请注意 `==` 和 `!=` 函数，它们是典型的辅助函数。对于很多类来说，`==` 和 `!=` 具有明显的意义，但它们又并不是对所有类都有意义，因此编译器无法像处理拷贝构造函数和赋值运算符那样为你定义默认的 `==` 和 `!=` 函数。

还请注意我们引入了一个辅助函数 `is_date()`。这个函数代替了 `Date::check()`，因为检查一个日期是否有效很大程度上与日期的描述是无关的。例如，我们无需知道日期对象是如何描述的，就可以判断“2008 年 1 月 30 日”是有效的，“2008 年 2 月 30 日”是无效的。还有一些日期相关的问题可能依赖于描述方法(例如，我们可以表示“1066 年 1 月 30 日”吗)，但这可以由 `Date` 的构造函数来处理(如果需要的话)。

9.8 Date 类

现在，让我们将这一章介绍的内容组合在一起，看看能设计出什么样的 `Date` 类来。下面代码中函数体都只是“...”的注释，因为具体实现很复杂(请不要现在就尝试实现它们)。首先，我们将

声明放在头文件 Chrono. h 中:

```
// file Chrono.h

namespace Chrono {

class Date {
public:
    enum Month {
        jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
    };

    class Invalid { };    // to throw as exception

    Date(int y, Month m, int d);    // check for valid date and initialize
    Date();                        // default constructor
    // the default copy operations are fine

    // nonmodifying operations:
    int day() const { return d; }
    Month month() const { return m; }
    int year() const { return y; }

    // modifying operations:
    void add_day(int n);
    void add_month(int n);
    void add_year(int n);
private:
    int y;
    Month m;
    int d;
};

bool is_date(int y, Date::Month m, int d);    // true for valid date

bool leapyear(int y);    // true if y is a leap year

bool operator==(const Date& a, const Date& b);
bool operator!=(const Date& a, const Date& b);

ostream& operator<<(ostream& os, const Date& d);

istream& operator>>(istream& is, Date& dd);

} // Chrono
```

定义在 Chrono. cpp 中:

```
// Chrono.cpp

namespace Chrono {

// member function definitions:

Date::Date(int yy, Month mm, int dd)
    : y(yy), m(mm), d(dd)
{
    if (!is_date(yy,mm,dd)) throw Invalid();
}

Date& default_date()
{
    static Date dd(2001,Date::jan,1);    // start of 21st century
    return dd;
}
```

```

}

Date::Date()
    :y(default_date().year()),
    m(default_date().month()),
    d(default_date().day())
{
}

void Date::add_day(int n)
{
    // ...
}

void Date::add_month(int n)
{
    // ...
}

void Date::add_year(int n)
{
    if (m==feb && d==29 && !leapyear(y+n)) { // beware of leap years!
        m = mar; // use March 1 instead of February 29
        d = 1;
    }
    y+=n;
}

// helper functions:

bool is_date(int y, Date::Month m, int d)
{
    // assume that y is valid

    if (d<=0) return false; // d must be positive

    int days_in_month = 31; // most months have 31 days

    switch (m) {
    case Date::feb: // the length of February varies
        days_in_month = (leapyear(y)) ? 29 : 28;
        break;
    case Date::apr: case Date::jun: case Date::sep: case Date::nov:
        days_in_month = 30; // the rest have 30 days
        break;
    }

    if (days_in_month < d) return false;

    return true;
}

bool leapyear(int y)
{
    // see exercise 10
}

bool operator==(const Date& a, const Date& b)
{
    return a.year()==b.year()
        && a.month()==b.month()
        && a.day()==b.day();
}

bool operator!=(const Date& a, const Date& b)
{
    return !(a==b);
}

ostream& operator<<(ostream& os, const Date& d)
{

```

```

        return os << '(' << d.year()
                << ',' << d.month()
                << ',' << d.day() << ')';
    }

    istream& operator>>(istream& is, Date& dd)
    {
        int y, m, d;
        char ch1, ch2, ch3, ch4;
        is >> ch1 >> y >> ch2 >> m >> ch3 >> d >> ch4;
        if (!is) return is;
        if (ch1!='(' || ch2!=',' || ch3!=',' || ch4!=')') { // oops: format error
            is.clear(ios_base::failbit); // set the fail bit
            return is;
        }

        return is;
    }

    enum Day {
        sunday, monday, tuesday, wednesday, thursday, friday, saturday
    };

    Day day_of_week(const Date& d)
    {
        // ...
    }

    Date next_Sunday(const Date& d)
    {
        // ...
    }

    Date next_weekday(const Date& d)
    {
        // ...
    }

} // Chrono

```

函数>>和<<的实现将在10.7节和10.8节中详细解释。



简单练习

本练习的目的就是使一系列 Date 版本能正常工作。对每个版本，请定义一个名为 today 的 Date 对象，初值为 1978 年 6 月 25 日。然后，定义一个名为 tomorrow 的 Date 对象，通过拷贝 today 对其赋值，随后使用 add_day() 将其向后推移一天。最后，使用 9.8 节中定义的 << 输出 today 和 tomorrow。

有效日期的检查可以简单实现。但应保证不接受 [1, 12] 范围之外的月份和 [1, 31] 范围之外的日期。对每个版本至少用一个无效日期来测试(如 2004 年 13 月 -5 日)。

1. 9.4.1 节中的版本。
2. 9.4.2 节中的版本。
3. 9.4.3 节中的版本。
4. 9.7.1 节中的版本。
5. 9.7.4 节中的版本。



思考题

1. 本章中所描述的类的两个组成部分是什么？

1. 在一个类中, 定义和实现的区别是什么?
2. 本章中最初定义的 `Date` 结构有什么局限和问题?
3. 为什么要为 `Date` 类型定义构造函数来取代函数 `init_day()`?
4. 什么是不变式? 给出一个例子。
5. 什么时候应该将函数定义置于类定义内? 什么时候又应该置于类外? 为什么?
6. 在程序中什么时候应该使用运算符重载? 给出一个你可能想重载的运算符列表(对于每一个请给出一个原因)。
7. 为什么应该令一个类的公有接口尽量小?
8. 为一个成员函数加上 `const` 限定符有什么作用?
9. 为什么辅助函数最好放在类定义之外?



术语

内置类型	枚举	不变式	class	枚举量	描述
const	辅助函数	struct	构造函数	实现	结构
析构函数	内联	用户自定义类型	enum	接口	有效状态



习题

1. 对 9.1 节中介绍的真实世界中的对象(如烤面包机)列出可能的操作。
2. 设计并实现一个保存 `(name, age)` 对的 `Name_pairs` 类, 其中 `name` 是一个 `string`, `age` 是一个 `double`。将值对描述为一个名为 `name` 的 `vector<string>` 成员和一个名为 `age` 的 `vector<double>` 成员。提供一个输入操作 `read_names()`, 能读入一个名字列表。提供一个 `read_ages()` 操作, 提示用户为每个名字输入一个年龄。提供一个 `print()` 操作, 按 `name` 向量的顺序打印 `(name[i], age[i])` 对(每行一个值对)。提供一个 `sort()` 操作, 将 `name` 向量按字典序排序, 并相应地重排 `age` 向量使之与 `name` 向量的新顺序相匹配。将所有“操作”实现为成员函数。测试这个类(当然, 在设计过程中尽早测试并多测试)。
3. 将 `Name_pairs::print()` 函数替换为(全局)运算符 `<<`, 并为 `Name_pair` 定义 `==` 和 `!=` 运算符。
4. 考察 8.4 节最后那个令人头痛的例子。给它加上适当的缩进和解释每个语法结构意义的注释。注意, 这个例子并未做任何有意义的事情, 它只是单纯地为了说明令人困惑的代码风格。
5. 此题和接下来的几道题要求你设计并实现一个 `Book` 类, 你可以设想这是图书馆软件系统的一部分。`Book` 类应包含表示 ISBN 号、书名、作者和版权日期以及表示是否已经借出的成员。创建能返回这些成员的值函数, 以及借书和还书的函数。对于输入 `Book` 对象的数据进行简单的有效性检查, 例如, 只接受 `n-n-x` 形式的 ISBN 号, 其中 `n` 是一个整数, `x` 是一个数字或一个字母。
6. 为 `Book` 类添加运算符。添加 `==` 运算符, 用于检查两本书的 ISBN 号是否相等。定义 `!=` 运算符, 比较 ISBN 号是否不等。定义 `<<`, 分行输出书名、作者和 ISBN 号。
7. 为 `Book` 类创建一个名为 `Genre` 的枚举类型, 用以区分书籍的类型: 小说(`fiction`)、非小说类文学作品(`nonfiction`)、期刊(`periodical`)、传记(`biography`)、儿童读物(`children`)。为每本书赋予一个 `Genre` 值, 适当修改 `Book` 的构造函数和成员函数。
8. 为图书馆创建一个 `Patron` 类, 包含读者姓名、图书证号及借阅费(如果欠费的话)。创建访问这些成员的函数和设定借书费的函数。定义一个辅助函数, 返回一个布尔值, 表示读者是否欠费。
9. 创建一个 `Library` 类, 包含一个 `Book` 向量和一个 `Patron` 向量。定义一个名为 `Transaction` 的 `struct`, 包含一个 `Book` 对象、一个 `Patron` 对象和一个本章中定义的 `Date` 对象, 表示借阅记录。在 `Library` 类中定义一个 `Transaction` 向量。定义向图书馆添加图书、添加读者以及借出书籍的函数。当一个读者借出一本书时, 保证 `Library` 对象中有此读者和这本书的记录, 否则报告错误。然后检查读者是否欠费, 如果欠费就报告一个错误, 否则创建一个 `Transaction` 对象, 将其放入 `Transaction` 向量中。定义一个返回包含所有欠费读者姓名的向量的函数。
10. 实现 9.8 节中的 `leapyear()`。
11. 为 `Date` 类设计并实现一组辅助函数, 如 `next_workday`(假定除周六和周日外都是工作日)和 `week_of_year`

(假定第1周是1月1日所在那周,每周的第1天是周日)。

12. 改变 Date 类的描述,用1970年1月1日(第0天)至今的天数表示日期,用一个 long 型成员保存此天数,重新实现 9.8 节中的函数。保证拒绝用这种方法无法表示的日期(第0天之前的日期也拒绝,即不允许负数天数)。
13. 设计并实现一个有理数类 Rational。一个有理数由两部分组成:分子和分母,如 $5/6$ (六分之五或近似为 0.83333)。如果需要的话,请查找有理数的定义。为 Rational 类定义实现赋值、加、减、乘、除及相等判定的运算符,并定义转换至 double 型值的函数。为什么人们需要使用 Rational 类?
14. 设计并实现一个 Money 类,能进行包含美元和美分、精确到美分的计算,使用四舍五入规则(大于等于 0.5 美分入,小于 0.5 美分舍)。用一个 long 型成员以美分值表示金额,但输入/输出采用美元和美分的形式,如 \$ 123.45。不必考虑金额值超出 long 型范围的情况。
15. 改进 Money 类,加入货币功能(货币类型通过构造函数参数给出)。能接受浮点型的初值,只要能用 long 型准确表示即可。不允许非法操作,如 $\text{Money} * \text{Money}$ 这种无意义的操作,但只要你提供了美元(USD)和丹麦克朗(DKK)之间的汇率就可以支持 $\text{USD}1.23 + \text{DKK}5.00$ 这种有意义的操作。
16. 给出一个例子,使用 Rational 进行计算得到的结果比使用 Money 更好。
17. 给出一个例子,使用 Rational 进行计算得到的结果比使用 double 类型更好。



附言

用户自定义类型是非常多的,比本章所介绍的多得多。用户自定义类型,特别是类,是 C++ 的核心,是很多高效设计技术的关键。在本书剩余部分中,大部分内容都是关于类的设计和使用的。一个类或者一组类,是我们用来在代码中表达思想的机制。本章主要介绍了类的语言技术细节,本书其他部分则关注如何用类优美地表达有用的思想。

第二部分 输入和输出

第 10 章 输入/输出流

“学习科学可以使我们远离愚昧。”

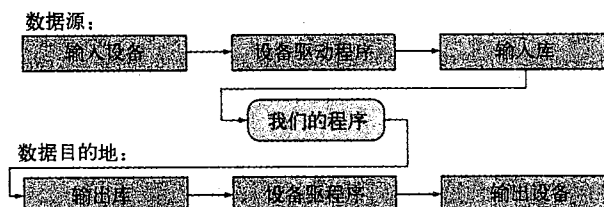
——Richard P. Feynman

在本章和第 11 章中，我们将从多个角度来学习 C++ 标准库中有关处理输入/输出的特性：输入/输出流(I/O stream)。我们将展示如何读写文件，如何处理输入/输出错误，如何进行格式化输入，以及如何为用户自定义类型提供输入/输出操作符及如何使用这类操作符。本章重点介绍基本问题：如何读写单个值，以及如何打开、读、写整个文件。最后会用一个例子来说明在一个较大规模的程序中应该考虑的与 I/O 相关的一些问题。第 11 章将会介绍更深入的技术细节。

10.1 输入和输出

如果没有数据，计算就毫无意义。我们需要将数据输入到程序中来进行一些有价值的计算，并将结果从中程序中取出。在 4.1 节中我们曾经提及，数据的输入源和输出目标非常广泛。如果我们不加小心，就会写出只能从特定的源输入数据，只能将结果输出到特定设备的程序。这对于某些特定应用，比如数码相机或引擎燃料喷射器传感器来说，是可以接受的（有时甚至是必需的）。但对于大多数应用，我们需要某种方法将程序的读写操作与实际进行输入/输出的设备分离开。如果我们必须直接访问每种设备，那么当有新的显示器或磁盘产品面市时，我们就必须修改程序，或者将用户局限于程序所支持的设备，这是很荒谬的。

大多数现代操作系统都将 I/O 设备的处理细节放在设备驱动程序中，通过一个 I/O 库访问设备驱动程序，这就使不同设备源的输入/输出尽可能地相似。一般地，设备驱动程序都位于操作系统较深的层次中，大多数用户是看不到它们的。I/O 库给出了输入/输出的一个抽象，从而程序员不必关心具体的设备和设备驱动程序：



如果操作系统使用这样一个模型,则所有输入和输出可以看做字节(字符)流,由输入/输出库处理。于是,我们程序员的工作就变为

- 1) 创建指向恰当数据源和数据目的地的 I/O 流。
- 2) 读和写这些流。

数据在程序和设备间实际是如何传输的这类细节,都是由 I/O 库和驱动程序来处理的。在本章和第 11 章中,我们将介绍如何使用 C++ 标准库来格式化 I/O 流中的数据。

从程序员的角度,输入和输出可以分为多种不同类型,如下所示:

- 大量数据项构成的流(这类流通常对应文件、网络连接、录音设备或显示设备)。
- 通过键盘来与用户交互的流。
- 通过图形界面与用户交互的流(输出对象、接收鼠标点击事件等)。

此分类法并不是唯一可能的分类,而且在这种分类中,三类 I/O 流的划分不是那么清晰。例如,如果一个输出字符流是一个以浏览器为目的地的 HTTP 文档,那么它看起来更像一个用户交互流,而且它可以包含图形元素。反过来,与 GUI(用户图形界面)交互的流也可能以一个字符序列的形式呈现给程序。这种分类法虽然不完美,但它很适合我们的工具:前两类 I/O 可以用 C++ 标准库中的 I/O 流实现,而且大多数操作系统都直接支持这两类 I/O。从第 1 章开始,我们就已经使用 `iostream` 库了,在本章和第 11 章中,我们仍主要关注这方面的内容。图形化输出和图形化用户交互则由其他一些库支持,我们将在第 12 章至第 16 章中讨论这部分内容。

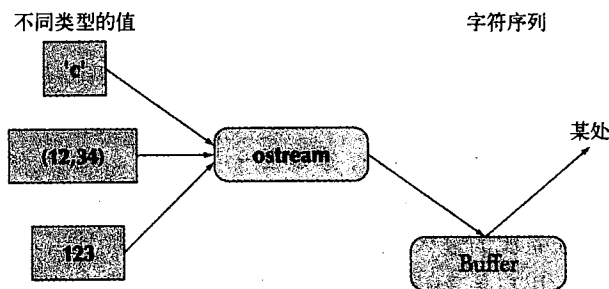
10.2 I/O 流模型

C++ 标准库提供了两个数据类型, `istream` 用于处理输入流, `ostream` 用于处理输出流。我们已经使用过标准输入流 `cin` 和标准输出流 `cout`, 因此我们已经了解了应该如何使用标准库中的这部分特性(通常称之为 `iostream` 库)。

一个 `ostream` 实现

- 将不同类型的值转换为字符序列。
- 将这些字符发送到“某处”(如控制台、文件、主存或者另外一台计算机)。

我们可以用下图来表示 `ostream`:

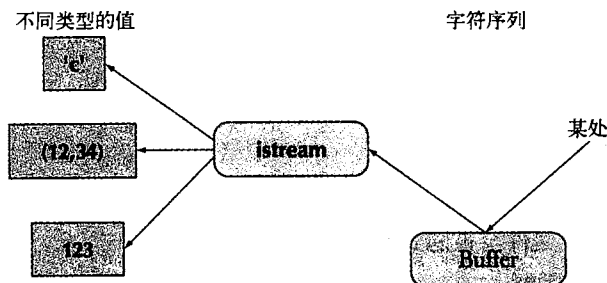


你提交给 `ostream` 的数据被保存在 `buffer` 数据结构中,通过它与操作系统通信。你可能会发现,当你将数据写入 `ostream`,可能一段“延迟”之后字符才出现在目的设备,这通常是因为字符还在缓冲区之中。缓冲技术是提高性能的重要技术,而处理大量数据时性能是很重要的。

一个 `istream` 实现

- 将字符序列转换为不同类型的值。
- 从某处(如控制台、文件、主存或另外一台计算机)获取字符。

一个 istream 可用下图描述:



与 ostream 一样, istream 也使用一个缓冲区来与操作系统通信。istream 的缓冲区很多情况下对用户是可见的。当你使用一个与键盘相关联的 istream 时, 你所键入的内容都留在缓冲区中, 直至你敲回车键为止(回车/换行), 你可以使用清除键(退格)来“改变你的主意”(直至敲回车键为止)。

输出的一个主要目的就是生成人类可读的数据形式。考虑 email 消息、学术论文、网页、账单记录、商务报告、通讯录、目录、设备状态信息等实例, 因此, ostream 提供了很多特性, 用于格式化文本以适应不同需求。同样为了易于人类阅读, 很多输入数据也是由人类编写或者格式化过了的。因此, istream 提供了一些特性, 用于读取由 ostream 生成的输出内容。我们将在 11.2 节介绍格式化输入/输出, 在 11.3.2 节中介绍如何读取非字符型输入数据。输入的复杂性很大程度上在于错误处理。为了能给出更为实际的例子, 我们将从数据文件相关的 istream 模型开始。

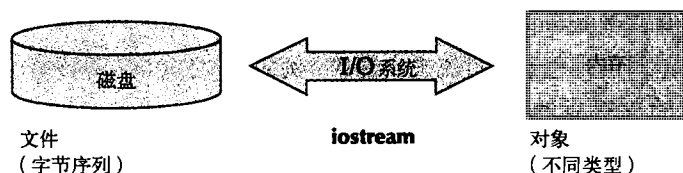
10.3 文件

通常, 我们需要处理的数据会多得难以放入计算机的内存之中, 因此我们将大部分数据存放于磁盘或其他大容量存储设备中。这种设备还具有另外一个我们需要的特性: 断电后, 保存在其中的数据不会丢失——即数据是持久的。在最基本的层面上, 一个文件可以简单地看做一个从 0 开始编号的字节序列:



通常每个文件都有自己的格式, 即有一组规则来确定其中字节的含义。例如, 如果是一个文本文件, 前 4 个字节就是文本的前 4 个字符。如果是一个二进制表示的整数文件, 同样是前 4 个字节, 表示的就是第 1 个整数了(参见 11.3.2 节)。格式之于磁盘文件的作用, 与类型之于内存对象的作用是一样的。当(且仅当)我们知道一个文件的格式(参见 11.2 ~ 11.3 节), 我们就能弄清文件中比特流的意思了。

对于一个文件, ostream 将内存中的对象转换为字节流, 再将字节流写到磁盘上。istream 进行相反的操作, 即它从磁盘获取字节流, 将其转换为对象:



多数情况下, 我们假定这些“磁盘上的字节”都是常用字符集中的字符。这个假设并不总是成立, 但绝大多数情况下我们可以认为它是对的, 而且, 其他的字符集表示方式也是不难处理的。我们还假定所有文件都保存在磁盘上(即保存在旋转磁存储设备上)。当然, 这个假设也不总是成立

(考虑闪存设备),但在这个层面上,实际采用什么存储设备对程序设计来说没什么区别,这也是文件和流抽象层的好处之一。

为了读取一个文件,我们需要

- 1) 知道文件名
- 2) (以读模式)打开文件
- 3) 读出字符
- 4) 关闭文件(虽然通常文件会被隐式地关闭)

为了写一个文件,我们需要

- 1) 知道文件名或者为新文件命名
- 2) (以写模式)打开文件或创建一个新文件
- 3) 写入我们的对象
- 4) 关闭文件(虽然通常文件会被隐式地关闭)

实际上我们之前已经掌握了基本的文件读写方法了,因为关联到一个文件的 `ostream` 对象的使用方式与 `cout` 是完全一样的, `istream` 对象则与 `cin` 完全一样,而对于 `cin` 和 `cout`,我们已经很熟悉了。我们将在 11.3.3 节中介绍一些只用于文件的操作。但现在,我们还是先了解如何打开文件,接下来关注那些可以用于所有 `ostream` 和 `istream` 对象的操作和技术。

10.4 打开文件

如果你要读或写一个文件,你需要打开一个与文件相关联的流。`ifstream` 是用于文件的 `istream` 流(读取文件), `ofstream` 是用于文件的 `ostream` 的流(写文件), `fstream` 是用于文件的 `iostream`,既可以写又可以读。文件流必须与某个文件相关联,然后才可使用。例如:

```
cout << "Please enter input file name: ";
string name;
cin >> name;
ifstream ist(name.c_str());    // ist is an input stream for the file named name
if (!ist) error("can't open input file ",name);
```

定义一个 `ifstream` 对象时,如果给出一个字符串作为参数,则以读模式打开以此为名的文件,并将它与 `ifstream` 对象相关联。函数 `c_str()` 是 `string` 类的一个成员函数,将 `string` 对象转换为更为低层的 C 风格字符串。许多系统接口函数的参数都要求这种类型的字符串。“`!ist`”用来检测文件是否正常打开了。如果已经成功打开,随后我们就可以像读取其他任何 `istream` 对象一样从文件中读取数据了。例如,假定已经对 `Point` 类定义了输入操作符 `>>`,我们可以像下面代码这样从文件中读取 `Point` 对象:

```
vector<Point> points;
Point p;
while (ist>>p) points.push_back(p);
```

与读文件的过程类似,写文件也是通过流来实现的,例如:

```
cout << "Please enter name of output file: ";
string oname;
cin >> oname;
ofstream ost(oname.c_str()); // ost is an output stream for a file named name
if (!ost) error("can't open output file ",oname);
```

用一个字符串参数定义一个 `ofstream` 对象,会打开以该字符串为名的文件与流对象相关联。“`!ost`”检测文件是否成功打开。如果成功打开,我们就可以像处理其他 `ostream` 对象一样向文件

中写入数据了,例如:

```
for (int i=0; i<points.size(); ++i)
    ost << '(' << points[i].x << ',' << points[i].y << ")\n";
```

当程序离开文件流对象的作用域时,对象会被销毁,它所关联的文件也会被关闭。当文件被关闭时,它关联的缓冲区会被刷新,即缓冲区中的字符会被写回文件。

一般来说,我们最好在较早的时间,在任何重要的计算都尚未开始之前就打开文件。毕竟,如果我们在完成计算之后才发现无法保存结果,将会是对计算资源的巨大浪费。

在创建 `ostream` 或 `istream` 对象时隐式打开文件,并依靠流对象的作用域来处理文件关闭,这是一个理想的方法。例如:

```
void fill_from_file(vector<Point>& points, string& name)
{
    ifstream ist(name.c_str());    // open file for reading
    if (!ist) error("can't open input file ",name);
    // ... use ist ...
    // the file is implicitly closed when we leave the function
}
```

你也可以通过 `open()` 和 `close()` 操作显式打开和关闭文件(参见附录 B.7.1)。但是,依靠作用域的方式最大限度地降低了两类错误出现的概率:在打开文件之前或关闭之后使用文件流对象。例如:

```
ifstream ifs;
// ...
ifs >> foo;                // won't succeed: no file opened for ifs
// ...
ifs.open(name,ios_base::in); // open file named name for reading
// ...
ifs.close();                // close file
// ...
ifs >> bar;                  // won't succeed: ifs' file was closed
// ...
```

在真实的程序中,通常这类错误更难以定位。幸运的是,如果尚未关闭一个文件就第二次打开它, `open()` 会返回一个错误,因此这种错误很容易发现。例如:

```
fstream fs;
fs.open("foo", ios_base::in);    // open for input
// close() missing
fs.open("foo", ios_base::out);    // won't succeed: ifs is already open
if (!fs) error("impossible");
```

因此,在打开一个文件之后,一定不要忘记检测是否成功了。

那么我们为什么还要用显式的 `open()` 和 `close()` 呢?是这样的,我们偶尔会遇到这样一种情况——使用文件的范围不能简单包含于任何流对象的作用域中,此时我们就不得不使用显式 `open()` 和 `close()` 了。但这种情况实在是太少见了,因此我们不必为此担心。更确切地说, `iostream`(以及 C++ 标准库的其他部分)都使用作用域的编程风格,只有在不使用这种风格的代码中才能找到上述情况。

在第 11 章中我们会看到,与文件相关的话题还有很多,但现在我们只要了解它们作为数据源和数据目的地的使用方法就足够了。如果我们假定用户必须直接键入所有输入数据的话,这样写出的程序会非常不实用。而使用文件,你在调试过程中就可以反复从文件读取输入。从程序员的角度来看,这也是文件的一个很大的优点。

10.5 读写文件

思考这样一个问题:你如何从一个文件中读取一些实验结果,并将其描述为内存数据对象?

这些实验结果可能是从气象站获取的温度数据：

```
0 60.7
1 60.6
2 60.3
3 59.22
...
```

这个数据文件包含了一个(小时, 温度)数值对的序列。小时的值从 0 到 23, 温度为华氏度值。没有更多的格式, 即这个文件不包含任何特殊的头信息(例如温度读数是从何获取的)、值的单位、标点(例如在每个值对外加上括号)或者终止符。这是一个最为简单的情形。

我们可以用一个 Reading 类型来描述温度读数：

```
struct Reading {           // a temperature reading
    int hour;              // hour after midnight [0:23]
    double temperature;    // in Fahrenheit
    Reading(int h, double t) : hour(h), temperature(t) {}
};
```

有了这样一个类型, 我们就可以读取温度读数了：

```
vector<Reading> temps;    // store the readings here
int hour;
double temperature;
while (ist >> hour >> temperature) {
    if (hour < 0 || 23 < hour) error("hour out of range");
    temps.push_back(Reading(hour, temperature));
}
```

这是一个典型的输入循环。正如前面介绍的, istream 流 ist 可以是一个输入文件流 (ifstream), 也可以是标准输入流 cin (的一个别名), 或者是任何其他类型的 istream。对于这段代码而言, 它并不关心这个 istream 到底是从哪里获取数据的。我们的程序所关心的只是: ist 是一个 istream, 而且数据格式如我们所期望的。下一节我们将讨论一个有趣的问题: 如何在输入数据中检测错误, 以及发现格式错误后我们应该如何做。

写文件通常比读文件要简单。再重复一遍, 一旦一个流对象已经被初始化, 我们不必了解它到底是哪种类型的流。特别地, 对于上一节介绍的输出文件流 (ofstream), 我们可以像使用其他任何 ostream 一样来使用它。例如, 我们可能想输出带括号的温度读数数值对：

```
for (int i=0; i<temps.size(); ++i)
    ost << '(' << temps[i].hour << ',' << temps[i].temperature << ")\n";
```

最终的程序就可以读取原始的温度读数文件, 然后利用上面的代码创建一个新文件, 其中每个数值对的格式为(小时, 温度)。

由于文件流对象在离开其作用域时会自动关闭所关联的文件, 因此完整的程序如下所示：

```
#include "std_lib_facilities.h"
```

```
struct Reading {           // a temperature reading
    int hour;              // hour after midnight [0:23]
    double temperature;    // in Fahrenheit
    Reading(int h, double t) : hour(h), temperature(t) {}
};

int main()
{
    cout << "Please enter input file name: ";
    string name;
    cin >> name;
    ifstream ist(name.c_str());    // ist reads from the file named "name"
    if (!ist) error("can't open input file ", name);
```

```

cout << "Please enter name of output file: ";
cin >> name;
ofstream ost(name.c_str()); // ost writes to a file named "name"
if (!ost) error("can't open output file ", name);

vector<Reading> temps; // store the readings here
int hour;
double temperature;
while (ist >> hour >> temperature) {
    if (hour < 0 || 23 < hour) error("hour out of range");
    temps.push_back(Reading(hour, temperature));
}

for (int i=0; i<temps.size(); ++i)
    ost << ' ' << temps[i].hour << ' '
        << temps[i].temperature << "\n";
}

```

10.6 I/O 错误处理

当处理输入时，我们必须预计到其中可能发生的错误并给出相应的处理措施。输入中会发生什么类型的错误呢？应该如何处理呢？输入错误可能是由于人的失误（错误理解了指令、打字错误、允许自家的小猫在键盘上散步等）、文件格式与规范不符、我们（程序员）错误估计了情况等原因造成的。发生输入错误的可能情况是无限的！但 `istream` 将所有可能的情况归结为 4 类，称为流状态（stream state）：

流状态	
<code>good()</code>	操作成功
<code>eof()</code>	到达输入末尾（“文件尾”）
<code>fail()</code>	发生某些意外情况
<code>bad()</code>	发生严重的意外

不幸的是，`fail()` 和 `bad()` 之间的区别并未准确定义，（定义新类型 I/O 操作的）程序员对其的观点各种各样。但是，基本的思想很简单：如果输入操作遇到一个简单的类型错误，则使流进入 `fail()` 状态，也就是假定你（输入操作的用户）可以从错误中恢复。另一方面，如果错误真的非常严重，比如发生了磁盘读故障，就应该让流进入 `bad()` 状态，也就是假定面对这种情况你所能做的很有限，只能退出输入。这种观点导致如下逻辑：

```

int i = 0;
cin >> i;
if (!cin) { // we get here (only) if an input operation failed
    if (cin.bad()) error("cin is bad"); // stream corrupted: let's get out of here!
    if (cin.eof()) {
        // no more input
        // this is often how we want a sequence of input operations to end
    }
    if (cin.fail()) { // stream encountered something unexpected
        cin.clear(); // make ready for more input
        // somehow recover
    }
}
}

```

!cin 可以读作“cin 不成功”或者“cin 发生了某些错误”或者“cin 的状态不是 `good()`”，这与“操作成功”正好相反。请注意我们在处理 `fail()` 时所使用的 `cin.clear()`，当流发生错误时，我们可以进

行错误恢复,为了恢复错误,我们显式地将流从 `fail()` 状态转移到其他状态,从而可以继续从中读取字符。`clear()` 就起到这样的作用——调用 `clear()` 后,流的状态就变为 `good()`。

下面是一个如何使用流状态的例子。假定我们要读取一个整数序列存入一个 `vector` 中,字符“*”或“文件尾”(在 Windows 平台是字符 `Ctrl + Z`, UNIX 平台是 `Ctrl + D`) 表示序列结束。例如:

1 2 3 4 5 *

上述功能可通过如下函数来实现:

```
void fill_vector(istream& ist, vector<int>& v, char terminator)
// read integers from ist into v until we reach eof() or terminator
{
    int i = 0;
    while (ist >> i) v.push_back(i);
    if (ist.eof()) return; // fine: we found the end of file

    if (ist.bad()) error("ist is bad"); // stream corrupted; let's get out of here!
    if (ist.fail()) { // clean up the mess as best we can and report the problem
        ist.clear(); // clear stream state,
                    // so that we can look for terminator
        char c;
        ist >> c; // read a character, hopefully terminator
        if (c != terminator) { // unexpected character
            ist.unget(); // put that character back
            ist.clear(ios_base::failbit); // set the state to fail()
        }
    }
}
```

注意,即使没有遇到终结符,函数也会返回。毕竟,我们可能已经读取了一些数据,而 `fill_vector()` 的调用者也许有能力从 `fail()` 状态中恢复过来。由于我们已经调用了 `clear()` 来清除状态,以便检查后续字符,因此,为了让调用者来处理 `fail()` 状态,我们必须将流状态重新置为 `fail()`。我们通过调用 `ist.clear(ios_base::failbit)` 来达到这一目的。对照简单的 `clear()`,带参数的用法有些人迷惑:当 `clear()` 调用带参数时,参数中所指出的 `istream` 状态位会被置位(进入相应状态),而未指出的状态位会被复位。通过将流状态设置为 `fail()`,表明遇到了一个格式错误,而不是一个更为严重的问题。我们用 `unget()` 将字符放回 `ist`,`fill_vector()` 的调用者可能需要使用此字符。`unget()` 函数是 `putback()` 的简洁版,它依赖流对象记住最后一个字符是什么,从而不需要在参数中明确给出。

如果 `fill_vector()` 的调用者希望知道是什么原因终止了输入,可以检测流是处于 `fail()` 还是 `eof()` 状态。当然也可以捕获 `error()` 抛出的 `runtime_error` 异常,但当 `istream` 处于 `bad()` 状态时,继续获取数据是不可能的,因此大多数调用者不必为此烦恼。这意味着,几乎在所有情况下,对于 `bad()` 状态,我们所能做的只是抛出一个异常。简单起见,可以让 `istream` 帮我们来。

```
// make ist throw if it goes bad
ist.exceptions(ist.exceptions() | ios_base::badbit);
```

语法看起来有些奇怪,但结果很简单,当此语句执行时,如果 `ist` 处于 `bad()` 状态,它会抛出一个标准库异常 `ios_base::failure`。在一个程序中,我们只能调用此 `exceptions()` 一次。这允许我们简化所有的输入过程,忽略对 `bad()` 的处理:

```
void fill_vector(istream& ist, vector<int>& v, char terminator)
// read integers from ist into v until we reach eof() or terminator
{
    int i = 0;
    while (ist >> i) v.push_back(i);
    if (ist.eof()) return; // fine: we found the end of file
```



```

// not good() and not bad() and not eof(), ist must be fail()
ist.clear(); // clear stream state
char c;
ist>>c;      // read a character, hopefully terminator

if (c != terminator) { // ouch: not the terminator, so we must fail
    ist.unget();        // maybe my caller can use that character
    ist.clear(ios_base::failbit); // set the state to fail()
}
}

```

这里使用了 `ios_base`，它是 `istream` 的一部分，包含常量（如 `badbit` 的定义）、异常（如 `failure` 的定义）以及其他一些有用的定义。你可以通过 `::` 操作符来使用它们，例如 `ios_base::badbit`（参见附录 B.7.2）。我们不想如此深入地讨论 `istream` 库的细节，要学习 `istream` 的所有内容，可能需要一门完整的课程。例如，`istream` 可以处理不同的字符集，实现不同的缓冲策略，还包含一些工具，能按不同语言的习惯格式化货币金额的输入/输出——我们曾经收到过一份错误报告，是关于乌克兰货币输入/输出格式的。如果需要的话，可以参考 Stroustrup 的《The C++ Programming Language》和 Langer 的《Standard C++ IOStreams and Locales》来钻研你想了解的 `istream` 库的内容。

与 `istream` 一样，`ostream` 也有 4 个状态：`good()`、`fail()`、`eof()` 和 `bad()`。不过，对于本书中的程序来说，输出错误要比输入错误罕见得多，因此我能通常不对 `ostream` 进行状态检测。如果程序运行环境中输出设备不可用、队列满或者发生故障的概率很高，我们就可以像处理输入操作那样，在每次输出操作之后都检测状态。

10.7 读取单个值

现在，我们已经知道如何读取以文件尾或者某个特定终结符结束的值序列了。我们还会学习一些更为复杂的例子，但在此之前，先看一个非常常见的应用模式：不断要求用户输入一个值，直至用户输入的值合乎要求为止。你可以从这个例子中学到几种常见的设计策略，我们通过“如何从用户获取一个合乎要求的值”这个问题的几种可选的解决方案来展示这些设计策略。我们从一个不那么令人满意的“初步尝试”方案开始，逐步提出改进方案，基本假设是：我们是在处理交互式的输入——程序给出提示信息，用户键入数据。假定要求用户输入一个 1 到 10 之间（包含 1 和 10）的整数：

```

cout << "Please enter an integer in the range 1 to 10 (inclusive):\n";
int n = 0;
while (cin>>n) { // read
    if (1<=n && n<=10) break; // check range
    cout << "Sorry "
        << n << " is not in the [1:10] range; please try again\n";
}

```

这段代码确实很丑陋，但它在某种程度上是可以正常工作的。如果你不喜欢使用 `break`（参见附录 A.6），可以将读操作和范围检查合并为一条语句：

```

cout << "Please enter an integer in the range 1 to 10 (inclusive):\n";
int n = 0;
while (cin>>n && !(1<=n && n<=10)) // read and check range
    cout << "Sorry "
        << n << " is not in the [1:10] range; please try again\n";

```

这不过是简单的改头换面，并未改变代码实质。为什么说这段代码只是“在某种程度上正常工作”呢？原因在于，这段代码只有在用户小心地输入整数的情况下才正常工作。如果用户键盘输入很

不熟练,本想输入6,但输入了t(在大多数键盘上,t恰好在6的上面),程序会不改变n的值就退出循环,于是n中的值就不在要求范围之内。这样的代码是不能被称为高质量代码的。而且,爱开玩笑的人(或者是勤奋的测试人员)还有可能从键盘键入“文件尾”符号(在Windows平台是Ctrl+Z,在UNIX平台是Ctrl+D)。于是,再次出现循环结束后n不在合法范围的情况。换句话说,为了获得可靠的输入,我们必须处理3个问题:

- 1) 用户输入超出范围的值。
- 2) 没有输入任何值(键入文件尾符号)。
- 3) 用户输入的内容类型错误(在本例中,未输入整数)。

我们想要如何应对这些问题呢?这也是程序设计过程中常常会遇到问题:我们真正想要的是什
么?在这里,对于每个错误,我们有3种可选的应对方式:

- 1) 在负责输入的代码中处理错误。
- 2) 抛出一个异常,让其他代码来处理这个错误(有可能终止程序)。
- 3) 忽略这个错误。

巧合的是,这恰恰是三种最为常用的错误处理策略,因此本例是展示如何处理错误的很好的例子。

人们很容易说,第三种策略(即忽略错误的方式)是不可接受的,但这样说有点过于居高临下了。如果我是在编写一个自己用的简单程序,还是可以随意选择实现策略的,包括“忘记”进行错误检测,不去管那些可能带来严重问题的计算结果。但是,如果是在编写一个将来可能长时间运行的程序,忽略这些错误就很愚蠢了。如果程序可能被他人所用的话,就更加不能在程序中忽略对这类错误的检测了。请注意,这里我有意识地使用了“我”而不是“我们”,因为“我们”可能会导致误解。也就是说,我们的观点是,如果超过两个人使用程序的话,第三种策略是不能接受的,但对于个人编写程序个人使用的情况,就不必那么绝对了。

在第一和第二种策略间进行取舍是很困难的。对于某个给定程序,可能有很好的理由选择两种策略中的任何一个。首先注意,在大多数程序中,对于用户不通过键盘进行输入的情况,还没有一种简洁的、局部性的方法来处理。因为当输入流关闭后,没有什么好的办法来请求用户输入一个数。我们当然可以重新打开cin(使用cin.clear()),但用户很可能不是意外地关闭输入流的。(你会意外地敲Ctrl+Z键吗?)如果一个程序希望读取一个整数,但却遇到一个“文件尾”,负责读入整数的代码最好放弃努力,寄希望于程序的其他部分能处理这个问题,也就是说,读取输入的代码应该抛出一个异常。这意味着我们本质上并不是要选择是抛出异常或是就地处理错误,而是要选择哪些错误我们应该就地处理。

10.7.1 将程序分解为易管理的子模块

下面我们尝试既处理超出范围的输入,又处理类型错误的输入:

```
cout << "Please enter an integer in the range 1 to 10 (inclusive):\n";
int n = 0;
while (true) {
    cin >> n;
    if (cin) { // we got an integer; now check it
        if (1 <= n && n <= 10) break;
        cout << "Sorry "
             << n << " is not in the [1:10] range; please try again\n";
    }
    else if (cin.fail()) { // we found something that wasn't an integer
        cin.clear(); // set the state back to good();
                    // we want to look at the characters
```

```

    cout << "Sorry, that was not a number; please try again\n";
    char ch;
    while (cin >> ch && !isdigit(ch)) ; // throw away non-digits
    if (!cin) error("no input"); // we didn't find a digit: give up
    cin.unget(); // put the digit back, so that we can read the number
}
else {
    error("no input"); // eof or bad: give up
}
}
// if we get here n is in [1:10]

```

这段代码又乱又冗长。它是如此之乱，以至于当有人需要编写让用户输入整数的程序时，我们绝不会建议他们这样写。但另一方面，我们确实要在代码中处理潜在的错误，因为用户确实时时在制造错误，我们该怎么办呢？这段程序如此之乱，是因为它把处理好几件不同事情的代码都混合在一起了：

- 读取数值
- 提示用户输入数值
- 输出错误信息
- 跳过“问题字符”
- 测试输入是否在所需范围内

一种常用的令代码更为清晰的方法是将逻辑上做不同事情的代码划分为独立的函数。例如，对于发现“问题字符”（如意料之外的字符）后进行错误恢复的代码，我们就可以将其分离出来：

```

void skip_to_int()
{
    if (cin.fail()) { // we found something that wasn't an integer
        cin.clear(); // we'd like to look at the characters
        char ch;
        while (cin >> ch) { // throw away non-digits
            if (isdigit(ch)) {
                cin.unget(); // put the digit back,
                            // so that we can read the number
            }
            return;
        }
    }
    error("no input"); // eof or bad: give up
}

```

已经有了上面的“工具函数”`skip_to_int()`后，代码就可以改写为：

```

cout << "Please enter an integer in the range 1 to 10 (inclusive):\n";
int n = 0;
while (true) {
    if (cin >> n) { // we got an integer; now check it
        if (1 <= n && n <= 10) break;
        cout << "Sorry " << n
            << " is not in the [1:10] range; please try again\n";
    }
    else {
        cout << "Sorry, that was not a number; please try again\n";
        skip_to_int();
    }
}
// if we get here n is in [1:10]

```

这段代码就好多了，但它还是太长、太乱，很难在程序中多次使用。需要进行大量的测试，才能保证其正确性。

我们到底需要什么样的操作呢？一个看起来挺合理的答案是：“我们需要一个读取任意一个整数的函数和一个读取一个指定范围内整数的函数”，如下所示：

```
int get_int();           // read an int from cin
int get_int(int low, int high); // read an int in [low:high] from cin
```

如果已有这些函数，我们至少能简单而又正确地使用它们。不难写出如下代码：

```
int get_int()
{
    int n = 0;
    while (true) {
        if (cin >> n) return n;
        cout << "Sorry, that was not a number; please try again\n";
        skip_to_int();
    }
}
```

`get_int()` 顽强地持续读入字符，直至发现可以解释为整数的数字符号为止。如果想让 `get_int()` 结束，必须输入一个整数或者键入文件尾符号（文件尾符号会使 `get_int()` 抛出一个异常）。

使用普通版本的 `get_int()`，我们可以写出具有范围检查功能的 `get_int()`：

```
int get_int(int low, int high)
{
    cout << "Please enter an integer in the range "
         << low << " to " << high << " (inclusive):\n";

    while (true) {
        int n = get_int();
        if (low <= n && n <= high) return n;
        cout << "Sorry "
             << n << " is not in the [" << low << ':' << high
             << "] range; please try again\n";
    }
}
```

这个版本的 `get_int()` 同样很顽强，它利用普通 `get_int()` 不断读取整数，直至读入的值在所需范围之内。

我们现在就可以像下面代码这样读取整数了：

```
int n = get_int(1,10);
cout << "n: " << n << endl;

int m = get_int(2,300);
cout << "m: " << m << endl;
```

不要忘记在某处捕获异常，这样，当 `get_int()` 确实不能读入一个整数时（虽然可能很罕见），我们就可以给出恰当的错误信息。

10.7.2 将人机对话从函数中分离

现在的 `get_int()` 函数还是混合着读取输入的代码和打印提示信息的代码。对于一个简单的程序来说，这可能没有什么问题。但在一个大型程序中，可能要打印不同的提示信息。例如，可能要想象下面这样调用 `get_int()`：

```
int strength = get_int(1,10, "enter strength", "Not in range, try again");
cout << "strength: " << strength << endl;
```

```
int altitude = get_int(0,50000,
    "Please enter altitude in feet",
    "Not in range, please try again");
cout << "altitude: " << altitude << "f above sea level\n";
```

一种可能的实现如下:

```
int get_int(int low, int high, const string& greeting, const string& sorry)
{
    cout << greeting << ": [" << low << ':' << high << "]\n";

    while (true) {
        int n = get_int();
        if (low <= n && n <= high) return n;
        cout << sorry << ": [" << low << ':' << high << "]\n";
    }
}
```

生成任意提示信息是很困难的,所以我们采取了“程式化”的方式。这通常就足够了,生成真正任意可变的提示信息,比如支持很多自然语言(如阿拉伯语、孟加拉语、汉语、丹麦语、英语以及法语),并不是一个初学者之力所能及的。

注意,我们的解决方案仍是不完整的:不进行范围检查的 `get_int()` 版本仍是个“多嘴的人”。这里所体现出的深层次的问题是:“工具函数”会在程序中很多地方被调用,因此不应该将提示信息“硬编码”到函数中。更进一步,库函数会在很多程序中被使用,也不应该向用户输出任何信息——毕竟,编写库的程序员甚至可能不知道使用库函数的程序所运行的计算机是否有人在操作,因此在库函数中输出信息有可能毫无意义。这也是为什么 `error()` 函数并没有简单地输出一条错误信息(参见 5.6.3 节),因为一般来说,我们无法知道向何处输出。

10.8 用户自定义输出操作符

为一个给定类型定义输出操作符 `<<` 是一件很简单的事情。要考虑的主要问题是不同人可能喜欢不同的输出形式,因此很难达成一个单一的格式。但即便无法提供一个令所有人都满意的单一输出格式,为用户自定义类型定义输出操作符 `<<` 通常也是一个好的策略。这样,我们至少可以在调试和早期开发期间,很容易地输出这个类型的对象。后期,我们可以提供一个更为复杂的 `<<`, 允许用户给出格式信息。而且,如果我们希望输出样式与 `<<` 提供的不同,可以简单地绕过 `<<`, 按照我们希望的格式直接输出用户自定义类型中的内容。

下面是为 9.8 节中 `Date` 类型定义的一个简单的输出操作符,它简单地打印年、月、日,用逗号分隔,两边加括号:

```
ostream& operator<<(ostream& os, const Date& d)
{
    return os << '(' << d.year()
        << ',' << d.month()
        << ',' << d.day() << ')';
}
```

这个输出操作符会将 2004 年 8 月 30 日打印为“(2004, 8, 30)”的形式。这种简单的成员列表的表示方式,对于成员数较少的类型来说很适合,除非我们有更好的想法或者更特殊的需求。

在 9.6 节中,我们提到,处理一个用户自定义运算符,实际是调用对应的函数。下面就是一个例子,假定已经为 `Date` 定义了上面的 `<<` 操作符,那么

```
cout << d1;
```

等价于(其中 `d1` 是 `Date` 类型的对象):

```
operator<<(cout,d1);
```

请注意 `operator<<()` 是如何接受一个 `ostream&` 作为第一个参数, 又将其作为返回值返回的。这就是为什么你可以将输出操作“链接”起来, 因为输出流按这种方式逐步传递下去了。例如, 你可以像下面这样输出两个日期:

```
cout<<d1<<d2;
```

这实际上是通过首先解析第一个 `<<`, 然后再解析第二个 `<<` 来实现的:

```
cout<<d1<<d2;    // means operator<<(cout,d1)<<d2;
                  // means operator<<(operator<<(cout,d1),d2);
```

也就是说, 连续输出两个对象 `d1` 和 `d2`, `d1` 的输出流是 `cout`, 而 `d2` 的输出流是第一个输出操作的返回结果。实际上, 为了输出 `d1` 和 `d2`, 上面代码中所示的三种写法中任何一种都是可以的。当然, 我们知道哪种最简洁、最易读。

10.9 用户自定义输入操作符

为一个给定类型和指定的输入格式定义输入操作符 `>>`, 关键在于错误处理, 因此可能会很棘手。

下面是为 9.8 节中 `Date` 类型定义的一个简单的输入操作符, 它要求的输入格式与上一节定义的 `<<` 的输出格式相同:

```
istream& operator>>(istream& is, Date& dd)
{
    int y, m, d;
    char ch1, ch2, ch3, ch4;
    is>>ch1>>y>>ch2>>m>>ch3>>d>>ch4;
    if (!is) return is;
    if (ch1!='(' || ch2!=',' || ch3!=',' || ch4!=')') { // oops: format error
        is.clear(ios_base::failbit);
        return is;
    }
    dd = Date(y, Date::Month(m), d); // update dd
    return is;
}
```

这个 `>>` 运算符读入形如“(2004, 8, 20)”的串, 并尝试用这三个整数创建一个 `Date` 对象。如往常一样, 输入比输出难处理得多。输入比输出更容易出错, 实际应用中也确实如此。

如果未发现“(整数, 整数, 整数)”格式的输入, `>>` 运算符会令流进入一个非正常状态(`fail`, `eof` 或 `bad`), 并且不会改变目标 `Date` 对象的值。成员函数 `clear()` 用来设置流的状态。显然, `ios_base::failbit` 将使流进入 `fail()` 状态。在输入故障的情况下保持目标 `Date` 对象不变是一个理想的策略, 这会使代码更干净。对于一个 `operator>>()` 来说, 理想目标是不读取或丢弃任何它未用到的字符, 但这太困难了: 因为我们可能必须读入大量字符, 才能捕获一个格式错误。例如, 考虑输入“(2004, 8, 30}”, 只有当读到最后的“}”时, 我们才能判断遇到了一个格式错误, 而一般来说, 指望退回这么多字符是靠不住的。唯一肯定可以保证的是用 `unget()` 退回一个字符。`operator>>()` 如果读入一个不合法的 `Date`, 如“(2004, 8, 32)”, `Date` 的构造函数会抛出一个异常, 这会使我们跳出函数 `operator>>()`。

10.10 一个标准的输入循环

在 10.5 节中, 我们学习了如何读写文件。但是, 随后我们就介绍更为深入的错误处理相关的

内容(参见 10.6 节), 因此输入循环还是最初的简单地读取一个文件, 从头读到尾的方式。这个假定是合理的, 因为我们通常对每个文件进行独立检查, 看其是否有效。但是, 我们通常是边读边检查的, 下面的代码给出了一个通用的解决策略, 假定 `ist` 是一个输入流:

```
My_type var;
while (ist>>var) {    // read until end of file
    // maybe check that var is valid
    // do something with var
}
// we can rarely recover from bad; don't try unless you really have to:
if (ist.bad()) error("bad input stream");
if (ist.fail()) {
    // was it an acceptable terminator?
}
// carry on: we found end of file
```

也就是说, 我们读入一组值, 将它们保存到变量中, 当无法再读入值的时候, 检查流的状态, 看看是什么原因造成的。类似 10.6 节, 我们可以稍稍改进一下这段代码, 令输入流在发生错误时抛出一个 `failure` 异常。这使我们可以避免不断检查故障。

```
// somewhere: make ist throw an exception if it goes bad:
ist.exceptions(ist.exceptions()|ios_base::badbit);
```

我们也可以指定一个字符作为终结符, 例如:

```
My_type var;
while (ist>>var) {    // read until end of file
    // maybe check that var is valid
    // do something with var
}
if (ist.fail()) {    // use '|' as terminator and/or separator
    ist.clear();
    char ch;
    if (!(ist>>ch && ch=='|')) error("bad termination of input");
}
// carry on: we found end of file or a terminator
```

如果我们不想要一个特别的终结符, 即只接受文件尾作为输入的终结, 可以简单地将 `error()` 调用之前的检测语句去掉。但是, 如果文件包含嵌套结构(例如, 文件由按月读数组成, 每月的读数是由每天的读数组成的, 而每天的读数是由每小时的读数组成的, 等等), 那么使用终结符是很有用的, 因此我们在后面的讨论中都假定使用终结符。

不幸的是, 这段代码仍然有些乱。特别是, 在读入很多文件的情况下, 重复检测终结符是很烦人的。我们可以定义一个函数来完成这部分功能:

```
// somewhere: make ist throw if it goes bad:
ist.exceptions(ist.exceptions()|ios_base::badbit);

void end_of_loop(istream& ist, char term, const string& message)
{
    if (ist.fail()) {    // use term as terminator and/or separator
        ist.clear();
        char ch;
        if (ist>>ch && ch==term) return;    // all is fine
        error(message);
    }
}
```

于是输入循环变为下面这样:

```

My_type var;
while (ist>>var) {    // read until end of file
    // maybe check that var is valid

    // do something with var
}
end_of_loop(ist,"bad termination of file");    // test if we can continue

// carry on: we found end of file or a terminator

```

函数 `end_of_loop()` 什么也不做, 除非流处于 `fail()` 状态。我们认为, 这样一个输入循环结构足够简单、足够通用, 适合很多应用。

10.11 读取结构化的文件

下面我们应用这个“标准输入循环”来解决一个实际问题。同样, 我们还是通过这个例子来展示有广泛应用范围的设计和编程技术。假定你要处理一个温度读数文件, 其结构为:

- 文件包含若干年份(包含按月的读数)
 - 年份以“{ year”开始, 后跟一个整数, 表示年份值, 例如 1900, 然后以“}”结束。
- 年份包含若干月份(包含按日的读数)
 - 月份以“{ month”开始, 后跟一个三字符月份名, 如 jan, 然后以“}”结束。
- 读数由一个时间和一个温度值组成
 - 读数以“(”开始, 后跟日期值, 小时值和温度值, 最后以“)”结束。

例如:

```

{ year 1990 }
{year 1991 { month jun }}
{ year 1992 { month jan ( 1 0 61.5) } {month feb (1 1 64) (2 2 65.2) }}
{year 2000
  { month feb (1 1 68) (2 3 66.66) ( 1 0 67.2)}
  {month dec (15 15 -9.2) (15 14 -8.8) (14 0 -2) }
}

```

这种格式有点怪, 通常结构化文件的格式都有些特别。现在工业界的发展趋势是, 结构化文件变得更有规律、更为层次化(如 HTML 和 XML 文件)。但现实情况是, 我们还是极少能控制需要处理的文件的格式。文件的格式就是这个样子, 我们要做的就是正确读取其内容。如果格式非常糟糕或文件包含太多错误, 我们可以编写一个格式转换程序, 将文件转换为更适合我们程序的格式。另一方面, 通常可以选择数据的内存表示形式来适应我们程序的需求, 因此我们通常可以选择合适的输出格式, 来满足特定的需求和偏好。

假定已经给定了上述的温度读数文件格式, 而我们只能接受它。幸运的是, 其中的年、月等组成部分都是可以自识别的(这有点像 HTML 或 XML 文件)。另一方面, 单个读数的格式对合法性检查没有什么帮助。例如, 没有什么信息可以帮助我们处理下列情况: 用户将日期值和小时值交换; 用户使用摄氏温度, 而程序期望的是华氏温度, 或者相反。我们必须应对这些情况。

10.11.1 内存表示

我们应该如何在内存中表示读数呢? 一个很直接的想法是定义 3 个类: Year、Month 和 Reading, 与输入准确匹配。Year 和 Month 在处理数据过程中显然很有用: 我们希望比较不同年份的温度, 计算每月的平均温度值, 比较一年中不同月份的温度, 比较不同年份相同月份的温度, 将温

度读数与日照记录和湿度读数进行匹配,等等。本质上说,Year 和 Month 与我们思考温度和天气的一般方式是吻合的:Month 包含了一个月的信息,而 Year 包含了一年的信息。但是 Reading 怎么样呢?它实际上与硬件(如传感器)的底层表示形式吻合。一个 Reading 对象的数据“(日期,小时,温度)”显得很奇怪,而且只在 Month 对象内才有意义。它还是非结构化的:读数不保证按日期或小时顺序给出。基本上,无论何时我们想对读数进行感兴趣的操作时,都要进行排序。

对于温度数据的内存表示,我们作如下假定:

- 如果获得了某月的任何一个读数,我们很可能会读取该月的其他很多读数。
- 如果获得了某日的一个读数,我们很可能会读取该日的很多读数。

如果情况确实如此,我们有必要将 Year 表示为包含 12 个 Month 的向量,Month 包含 30 个 Day 的向量,而 Day 包含 24 个温度值(每小时一个)。对于很多应用来说,这种方式简单、易于处理。因此,Day、Month 和 Year 都是简单数据结构,只是带有构造函数。由于我们计划在获取温度读数之前就创建 Month 和 Day,作为 Year 的一部分,因此我们需要一个“非读数”的概念,来表示某个小时数据还未读入。

```
const int not_a_reading = -7777; // less than absolute zero
```

类似地,我们引入“非月份”的概念来直接表示未读入数据的月份,以免不得不搜索该月所有日期来确定不包含的数据:

```
const int not_a_month = -1;
```

于是 3 个关键的类可以定义如下:

```
struct Day {
    vector<double> hour;
    Day(); // initialize hours to "not a reading"
};

Day::Day()
: hour(24)
{
    for (int i = 0; i < hour.size(); ++i) hour[i] = not_a_reading;
}

struct Month { // a month of temperature readings
    int month; // [0:11] January is 0
    vector<Day> day; // [1:31] one vector of readings per day
    Month() // at most 31 days in a month (day[0] wasted)
        : month(not_a_month), day(32) {}
};

struct Year { // a year of temperature readings, organized by month
    int year; // positive == A.D.
    vector<Month> month; // [0:11] January is 0
    Year() : month(12) {} // 12 months in a year
};
```

每个类本质上是一个简单向量,而 Month 和 Year 各有一个成员 month 和 year,用来表示月份和年份。

这里用到了好几个“魔数”(例如 24、32 和 12)。我们试图避免在程序中使用这种文字常量。这里使用的几个常量都是非常基本的(一年有几个月几乎是不会改变的),而且不会在程序其他部分使用。但是,我们保留它们并不是因为合理,而主要是为了能够提醒你“魔数”所存在的问题,符号常量几乎永远都是更好的选择(参见 7.6.1 节)。使用 32 作为一个月中的天数,肯定要进行合理的解释,此处,32 显然就是“有魔力的”。

10.11.2 读取结构化的值

类 `Reading` 值用来读取输入，因而很简单：

```
struct Reading {
    int day;
    int hour;
    double temperature;
};

istream& operator>>(istream& is, Reading& r)
// read a temperature reading from is into r
// format: ( 3 4 9.7 )
// check format, but don't bother with data validity
{
    char ch1;
    if (is>>ch1 && ch1!='(') { // could it be a Reading?
        is.unget();
        is.clear(ios_base::failbit);
        return is;
    }

    char ch2;
    int d;
    int h;
    double t;
    is>>d>>h>>t>>ch2;
    if (!is || ch2!=')') error("bad reading"); // messed-up reading
    r.day = d;
    r.hour = h;
    r.temperature = t;
    return is;
}
```

基本上，我们还是先检查格式是否合法，如果不合法，我们将文件状态置为 `fail()` 并返回。这允许我们尝试通过其他方式读取信息。另一方面，如果在读取了一些数据后才发现格式错误，就没有了错误恢复的机会，我们只能通过 `error()` 退出。

`Month` 的实现大致一样，只有一点不同：必须读入任意数目的 `Reading` 对象，而不是像 `Reading` 的 `>>` 那样只需读取一组固定个数的值：

```
istream& operator>>(istream& is, Month& m)
// read a month from is into m
// format: { month feb ... }
{
    char ch = 0;
    if (is>>ch && ch!='{') {
        is.unget();
        is.clear(ios_base::failbit); // we failed to read a Month
        return is;
    }

    string month_marker;
    string mm;
    is>>month_marker>>mm;
    if (!is || month_marker!="month") error("bad start of month");
    m.month = month_to_int(mm);

    Reading r;
```

```

int duplicates = 0;
int invalids = 0;
while (is >> r) {
    if (is_valid(r)) {
        if (m.day[r.day].hour[r.hour] != not_a_reading)
            ++duplicates;
        m.day[r.day].hour[r.hour] = r.temperature;
    }
    else
        ++invalids;
}
if (invalids) error("invalid readings in month", invalids);
if (duplicates) error("duplicate readings in month", duplicates);
end_of_loop(is, '}', "bad end of month");
return is;
}

```

我们随后再来讨论 `month_to_int()`，它将月份的符号表示(如 `jun`)转换为一个 0 到 11 之间的整数值。注意，代码中使用了 10.10 节中给出的 `end_of_loop()` 来检测终结符。我们对不合法的和重复的 `Reading` 进行计数，这对有的人可能会有用。

`Month` 的 `>>` 会快速检查 `Reading` 对象的合法性，然后将其存入向量：

```

const int implausible_min = -200;
const int implausible_max = 200;

bool is_valid(const Reading& r)
// a rough test
{
    if (r.day < 1 || 31 < r.day) return false;
    if (r.hour < 0 || 23 < r.hour) return false;
    if (r.temperature < implausible_min || implausible_max < r.temperature)
        return false;
    return true;
}

```

最后，我们可以设计 `Year` 的输入函数，与 `Month` 的类似：

```

istream& operator>>(istream& is, Year& y)
// read a year from is into y
// format: { year 1972 ... }
{
    char ch;
    is >> ch;
    if (ch != '{') {
        is.unget();
        is.clear(ios::failbit);
        return is;
    }

    string year_marker;
    int yy;
    is >> year_marker >> yy;
    if (!is || year_marker != "year") error("bad start of year");
    y.year = yy;

    while(true) {
        Month m; // get a clean m each time around
        if (!(is >> m)) break;
        y.month[m.month] = m;
    }

    end_of_loop(is, '}', "bad end of year");
    return is;
}

```

我们当然想将“烦人的相似”变成就是“相似”，这样就可以将这几段代码合而为一了，但其中有一个重要的不同。请看输入循环，下面的代码是你所期待的吗？

```
Month m;
while (is >> m)
    y.month[m.month] = m;
```

也许是，因为这就是目前为止我们设计输入循环的方法，但它是错的。问题在于 operator >> (istream& is, Month& m) 并不为 m 赋予新值，而只是简单地将 Reading 中的数据填入 m。因此，反复执行 is >> m 会不断将数据填入唯一的一个 Month 对象 m。糟糕！每一个月份实际上都从同年之前月份继承了所有读数，而没有清空该月不包含的读数。因此，每次执行 is >> m 时，我们需要一个崭新的、干净的 Month 对象。最简单的方法是将 m 的定义移入循环内，这样每次执行 is >> m 前都会对其初始化。另一种方式是令 operator >> (istream& is, Month& m) 在读入数据之前将 m 置空，或者让输入循环完成这一工作。

```
Month m;
while (is >> m) {
    y.month[m.month] = m;
    m = Month(); // "reinitialize" m
}
```

试试下面的程序：

```
// open an input file:
cout << "Please enter input file name\n";
string name;
cin >> name;
ifstream ifs(name.c_str());
if (!ifs) error("can't open input file",name);

ifs.exceptions(ifs.exceptions()|ios_base::badbit); // throw for bad()

// open an output file:
cout << "Please enter output file name\n";
cin >> name;
ofstream ofs(name.c_str());
if (!ofs) error("can't open output file",name);

// read an arbitrary number of years:
vector<Year> ys;
while(true) {
    Year y; // get a freshly initialized Year each time around
    if (!ifs>>y) break;
    ys.push_back(y);
}
cout << "read " << ys.size() << " years of readings\n";

for (int i = 0; i<ys.size(); ++i) print_year(ofs,ys[i]);
```

我们把 print_year() 的实现作为练习。

10.11.3 改变表示方法

为了使 Month 的 >> 能正常工作，我们需要提供一个方法，能读入月份的符号表示，并转换为 0 到 11 之间的整数值。一种冗长的表示方法是使用 if 语句：

```
if (s=="jan")
    m = 1;
else if (s=="feb")
    m = 2;
...
```

这种方法不仅冗长，而且将月份名硬编码到了程序中。更好的方法是将月份名存入一个表中，使得即便我们不得不改变符号表示时，也无需改动主程序。我们决定用一个向量 `vector<string>` 来描述月份的符号表示，另外设计一个初始化函数和一个查找函数：

```
vector<string> month_input_tbl; // month_input_tbl[0]=="jan"
```

```
void init_input_tbl(vector<string>& tbl)
```

```
// initialize vector of input representations
```

```
{
    tbl.push_back("jan");
    tbl.push_back("feb");
    tbl.push_back("mar");
    tbl.push_back("apr");
    tbl.push_back("may");
    tbl.push_back("jun");
    tbl.push_back("jul");
    tbl.push_back("aug");
    tbl.push_back("sep");
    tbl.push_back("oct");
    tbl.push_back("nov");
    tbl.push_back("dec");
}
```

```
int month_to_int(string s)
```

```
// is s the name of a month? If so return its index [0:11] otherwise -1
```

```
{
    for (int i=0; i<12; ++i) if (month_input_tbl[i]==s) return i;
    return -1;
}
```

免得你疑惑：C++ 标准库提供了一种简单的方法来完成相同的工作，参见 21.6.1 节 `map<string, int>` 的相关内容。

当需要进行输出时，我们将面临一个相反的问题。我们在内存中用一个整数表示月份，但输出时希望用符号表示的形式。解决方案与输入基本相似，只是把 `string` 到 `int` 的映射表变为 `int` 到 `string` 的映射表：

```
vector<string> month_print_tbl; // month_print_tbl[0]=="January"
```

```
void init_print_tbl(vector<string>& tbl)
```

```
// initialize vector of output representations
```

```
{
    tbl.push_back("January");
    tbl.push_back("February");
    tbl.push_back("March");
    tbl.push_back("April");
    tbl.push_back("May");
    tbl.push_back("June");
    tbl.push_back("July");
    tbl.push_back("August");
    tbl.push_back("September");
    tbl.push_back("October");
    tbl.push_back("November");
    tbl.push_back("December");
}
```

```
string int_to_month(int i)
```

```
// months [0:11]
```

```
{
    if (i<0 || 12<=i) error("bad month index");
    return month_print_tbl[i];
}
```

我们需要在某处调用初始化函数，比如在主函数 `main()` 的开始处：

```
// first initialize representation tables:
init_print_tbl(month_print_tbl);
init_input_tbl(month_input_tbl);
```

好了，你是否真正阅读了全部代码和注释了呢？还是眼睛一眨就跳到末尾了？记住，学习编写高质量代码的最好途径就是阅读大量代码。不管你信不信，本例中我们使用的方法很简单，但在没有帮助的情况下领悟其精髓并不容易。读取数据是很基本的，正确编写输入循环（正确初始化用到的变量）是很基本的，转换表示方式也很基本。也就是说，你应该学会这些。唯一的问题是，你是否能学会很好地使用这些技术，以及是否会错过太多以至于不能学好这些基本技术。

简单练习

1. 开始编写一个程序平面中的点，使用 10.4 节中所讨论的技术，首先定义包含两个表示坐标的成员 `x` 和 `y` 的数据类型 `Point`。
2. 借助 10.4 节中给出的代码和讨论的技术，提示用户输入 7 个 `(x, y)` 值对。当用户输入数据时，将其保存在一个名为 `original_points` 的向量中。
3. 打印 `original_points` 中的数据，看看结果如何。
4. 打开一个 `ofstream`，将每个点输出到名为 `mydata.txt` 的文件。在 Windows 平台上，我们建议使用 `.txt` 后缀，这样使用传统的文本编辑器（如写字板）可以很容易地查看文件中的数据。
5. 关闭 `ofstream`，然后打开一个 `ifstream`，与 `mydata.txt` 关联。从 `mydata.txt` 中读取数据，保存在一个名为 `processed_points` 的新的向量中。
6. 打印两个向量中的数据元素。
7. 比较两个向量，如果发现元素数目或值不符，打印“Something's wrong!”。

思考题

1. 对于大多数现代计算机系统，处理输入和输出时，要处理的设备种类有哪些？
2. `istream` 的基本功能是什么？
3. `ostream` 的基本功能是什么？
4. 从本质上看，文件是什么？
5. 什么是文件格式？
6. 给出 4 种需要进行 I/O 的设备类型。
7. 读取一个文件的 4 个步骤是什么？
8. 写一个文件的 4 个步骤是什么？
9. 给出 4 种流状态的名称和定义。
10. 讨论如何解决如下输入问题：
 - a) 用户输入了要求范围之外的值。
 - b) 未读到值（到达文件末尾）。
 - c) 用户输入了错误类型的数据。
11. 输入通常在哪些方面比输出困难？
12. 输出通常在哪些方面比输入困难？
13. 我们为什么通常希望将输入/输出与计算分离？
14. `istream` 的成员函数 `clear()` 最常用的两个用途是什么？
15. 对于一个用户自定义类型 `X`，`<<` 和 `>>` 通常的函数声明形式如何？

术语

<code>bad()</code>	<code>good()</code>	<code>ostream</code>	<code>buffer</code>	<code>ifstream</code>	输出设备
<code>clear()</code>	输入设备	输出运算符	<code>close()</code>	输入运算符	流状态
设备驱动程序	<code>istream</code>	结构化文件	<code>eof()</code>	<code>istream</code>	终结符

fail() ofstream unget() 文件 open()



习题

1. 一个文件中保存以空白符间隔的整数, 编写程序求此文件中所有整数之和。
2. 编写程序, 创建一个温度读数文件, 数据格式为 Reading 类型, Reading 的定义如 10.5 节所述, 向文件中填入至少 50 个温度读数。将此程序命名为 store_temps.cpp, 读数文件命名为 raw_temps.txt。
3. 编写程序, 从习题 2 创建的 raw_temps.txt 中读取数据, 存入一个向量, 随后计算数据集中温度的均值和中间值。将此程序命名为 temp_stats.cpp。
4. 修改习题 2 中的程序 store_temps.cpp, 为每个读数附加一个后缀 c 或者后缀 f, 分别表示摄氏温度和华氏温度。然后修改程序 temp_stats.cpp, 检测每个温度读数, 在存入向量之前将摄氏温度转换为华氏温度。
5. 编写 10.11.2 节中提到的 print_year() 函数。
6. 定义 Roman_int 类, 保存罗马数字(以 int 类型保存), 为其定义 << 和 >> 运算符。为其定义 as_int() 成员函数, 返回 int 型值, 使得对于 Roman_int 对象, 可以写出语句 cout << "Roman" << r << "equals" << r.as_int() << '\n';。
7. 修改第 7 章中的计算器程序, 使其接受罗马数字而不是阿拉伯数字, 例如 XXI + CIV == CXXV。
8. 编写程序, 接受两个文件名, 生成一个新文件, 内容为两个输入文件的拼接。
9. 两个文件包含已排序的、空白符间隔的单词, 编写程序将它们合并, 结果文件中单词仍有序排列。
10. 改写第 7 章中的计算器程序, 增加 "from x" 命令, 使其从文件 x 获取输入。增加一个 "to y" 命令, 实现输出到文件 y(包括计算结果和错误信息)。设计一系列的测试用例, 设计思路如 7.3 节所述, 用它们来测试改写后的计算器程序。讨论如何将这两个命令用于计算器程序的测试。
11. 编写程序, 对于一个文本文件, 找出其中空白符间隔的所有整数, 求它们的和。例如, "bear: 17 elephants 9 end" 应该输出 26。
12. 编写程序, 对于给定的一个文件名和一个单词, 输出文件中包含这个单词的所有行以及行号。提示: getline()。



附言

很多计算过程都包含将大量数据从某处移动到另一处的操作, 例如, 将文件中的文本复制到屏幕, 或者将音乐从计算机移动到 MP3 播放器。通常, 在迁移过程中还伴随着数据转换。如果数据可以看做值的序列(流), iostream 库很适合处理这类任务。在一般的编程工作中, 输入/输出部分的编码令人吃惊地占据了非常大的比例。这一方面是因为我们(或我们的程序)需要大量数据, 另一方面是数据进入程序的入口正是错误多发地带。因此, 我们必须努力使 I/O 部分更为简单, 并尽量降低有害数据“溜进”程序的几率。

第 11 章 定制输入/输出

“保持简洁：尽可能地简洁，但不要过度简单化。”

——Albert Einstein

在本章中，我们着重介绍如何采用第 10 章所提出的通用 `iostream` 框架来解决特定的输入/输出需求和偏好。其中涉及大量较为困难的细节，这一方面是由人类对输入内容的感觉所决定，另一方面则是由使用文件的实际应用限制所决定的。本章的最后一个例子会展示一个输入流的设计，它允许指定间隔符集合。

11.1 有规律的和无规律的输入和输出

`iostream` 库，也就是 ISO C++ 标准库的输入/输出部分，为文本(text)的输入和输出提供了一个统一的、可扩展的框架。这里的“文本”指的是任何可以表示为字符序列的数据。这样，当我们讨论输入/输出时，我们可以将 1234 这样的整数也看做文本，因为它可以写成 4 个字符 1、2、3 和 4。

到目前为止，我们对所有的输入源都是以相同方式处理的。但有时这是不够的，例如，有些文件可能与其他输入源不同，在其中我们可以定位单个字节(如通信连接)。类似地，我们还假定输入/输出格式完全由对象类型所决定，这也不完全正确，某些情况下这也是不够的。例如，我们常常会指定浮点数输出的数字个数(精度)。本章会提出一些方法，使我们可以按需求定制输入/输出。

作为程序员，我们更喜欢有规律的输入/输出：一致地处理所有内存对象；以相同的方式处理所有输入源；以及强制一个单一标准，限定对象进入和离开系统的表示方式。这样，我们就容易写出干净、简单、更易于维护而且通常更高效的代码。但是，程序的存在是为了服务于人类的，而人类都有自己强烈的偏好。因此，作为程序员，我们必须力争在程序复杂性和满足用户个人偏好间取得平衡。

11.2 格式化输出

人们常常会在意很多输出中的微小细节。例如，对一位物理学家来说，1.25(舍入到小数点后两位数字)与 1.24670477 可能是有很大不同的。而对于一位会计，“(1.25)”从法律角度看与“(1.2467)”是不一样的，而与“1.25”则是根本不同的(在金融文件中，括号有时表示亏损，也就是负值)。作为程序员，我们的目标是使我们的输出尽可能地清晰、尽可能地接近“客户”的期望。输出流(`ostream`)提供了很多方法格式化内置类型的输出。对于用户自定义类型，则需要由程序员定义适合的 `<<` 操作符。

对于输出，似乎有数不清的细节、优化的余地和不同的选择需要考虑；对于输入，要考虑的类似问题也不少。典型的例子有：用来表示小数点的字符(通常是点或逗号)；输出金额的方式；输出单词 `true`(或 `vrai` 或 `sandt`)而不是数值 1 来表示真；处理非 ASCII 字符集(如 Unicode)的方式；以及限制读入字符串的字符数目等。除非你需要使用这些功能，否则它们看起来很无趣。因此，我们将这些内容放在手册和专门的书籍中(如 Langer 的《Standard C++ IOSTreams and Locales》、Stroustrup 的《The C++ Programming Language》的第 21 章以及《ISO C++ 标准》的第 22 和 27 章)。

本书只介绍一些最常用的功能和一些一般性概念。

11.2.1 输出整数

整数值可以输出为八进制(octal, 基底为 8 的数制系统)、十进制(decimal, 人类常用的数制系统, 基底为 10)和十六进制(hexadecimal, 基底为 16 的数制系统)。如果你不了解这些数制, 请参考附录 A.2.1.1, 然后再继续学习本章。大多数输出都使用十进制。十六进制多用于输出硬件相关的信息, 原因在于一个十六进制数字精确地表示了 4 位二进制值。因此, 两个十六进制数字可以用来表示一个 8 位字节, 4 个十六进制数字表示 2 字节的值(通常称为半字), 8 个十六进制数字则表示 4 字节的值(通常是一个字或一个寄存器的大小)。当 20 世纪 70 年代 C 语言(C++ 的祖先)最初发明之时, 表示二进制位更多采用八进制, 现在则很少使用了。

我们可以指定(十进制)数 1234 以十进制、十六进制(通常简称为“hex”)或八进制输出:

```
cout << 1234 << "\t(decimal)\n"
    << hex << 1234 << "\t(hexadecimal)\n"
    << oct << 1234 << "\t(octal)\n";
```

字符‘\t’是“制表符”(tabulation character, 简称“tab”), 这段代码会输出如下内容:

```
1234 (decimal)
4d2 (hexadecimal)
2322 (octal)
```

符号 << hex 和 << oct 并不输出任何内容。前者通知流任何后来的整数值应该以十六进制输出, 而后者通知流以八进制输出后来的整数。例如:

```
cout << 1234 << '\t' << hex << 1234 << '\t' << oct << 1234 << '\n';
cout << 1234 << '\n';           // the octal base is still in effect
```

这段代码会输出:

```
1234 4d2 2322
2322           // integers will continue to show as octal until changed
```

注意, 最后一行的输出是一个八进制数。也就是说, oct、hex 和 dec(十进制输出)是持久的(不变的)——后来的整数一直按照这种数制输出, 直至我们指定新的数制。hex 和 oct 这种用来改变流的行为的关键字称为操纵符(manipulator)。

试一试 以十进制、十六进制和八进制输出你的出生年份。为每个值加上标识, 将

所有值显示在一行上, 利用制表符调整值的列位置。然后再输出你的年龄。

一个数值以非十进制显示, 总是让人看着有些迷惑。例如, 除非我们告诉你, 否则你一定会假定 11 表示(十进制)数值 11, 而不是 9(八进制数 11 所表示的十进制数值), 或者 17(十六进制数 11 所表示的十进制数值)。为了缓解这个问题, 我们可以要求 ostream 显示每个整数的基底。例如:

```
cout << 1234 << '\t' << hex << 1234 << '\t' << oct << 1234 << '\n';
cout << showbase << dec;           // show bases
cout << 1234 << '\t' << hex << 1234 << '\t' << oct << 1234 << '\n';
```

会输出:

```
1234 4d2 2322
1234 0x4d2 02322
```

于是, 十进制数将没有前缀, 八进制数将带前缀 0, 而十六进制数将带前缀 0x(或 0X)。这与 C++ 源程序中的整数文字常量的表示方式是完全一致的。例如:

```
cout << 1234 << '\t' << 0x4d2 << '\t' << 02322 << '\n';
```

如果是十进制输出格式, 这段代码会输出:

```
1234 1234 1234
```


会输出：

```
1234.57          (general)
1234.567890      (fixed)
1.234568e+003    (scientific)
```

操纵符 `fixed` 和 `scientific` 用来选择浮点数格式。奇怪的是，标准库没有提供 `general` 操纵符来设置默认格式。不过，我们可以自己定义一个，就像 `std_lib_facilities.h` 中所做的一样。当然，这需要对 `iostream` 库的内部工作机制有所了解：

```
inline ios_base& general(ios_base& b) // to complement fixed and scientific
// clear all floating-point format flags
{
    b.setf(ios_base::fmtflags(0), ios_base::floatfield);
    return b;
}
```

现在，我们可以写如下代码：

```
cout << 1234.56789 << '\t'
    << fixed << 1234.56789 << '\t'
    << scientific << 1234.56789 << '\n';
cout << 1234.56789 << '\n';           // floating format "sticks"
cout << general << 1234.56789 << '\t' // warning: general isn't standard
    << fixed << 1234.56789 << '\t'
    << scientific << 1234.56789 << '\n';
```

会得到如下输出：

```
1234.57      1234.567890  1.234568e+003
1.234568e+003                               // scientific manipulator "sticks"
1234.57      1234.567890  1.234568e+003
```

总之，基本的浮点数输出格式操纵符小结如下：

浮点数输出操纵符	
<code>fixed</code>	使用定点表示
<code>scientific</code>	使用尾数和指数表示方式。尾数总在 $[1: 10)$ 之间，也就是说，在小数点之前有单个非 0 数字
<code>general</code>	在 <code>general</code> 格式的精度下，自动选择 <code>fixed</code> 和 <code>scientific</code> 两者中更为精确的一种表示形式。 <code>general</code> 是默认格式，但如果想显式设定，你需要自己定义一个 <code>general()</code>

11.2.4 精度

默认设置下，`general` 格式用总共 6 位数字来输出一个浮点值。流会选择最适合的格式，浮点值按 6 位数字（`general` 格式的默认精度）所能表示的最佳近似方式进行舍入。例如：

1234.567 会输出 1234.57

1.2345678 会输出 1.23457

舍入规则采用常用的 4/5 规则：0 到 4 舍，5 到 9 入。注意，浮点格式只对浮点数起作用，于是

1234567 输出 1234567（因为这是一个整数）

1234567.0 输出 1.23457e+006

在第二个例子中，`ostream` 判断出 1234567.0 在 `fixed` 格式下不能只用 6 位数字输出，因此选择 `scientific` 格式，保持最为精确的表示形式。基本上，`general` 格式在 `scientific` 和 `fixed` 两种格式间进行选择，期望将浮点数最精确的表示形式呈现给用户，所采用的精度限定为 `general` 格式的精度——默认为 6 位数字长度。

试一试 编写代码输出浮点数 1234567.89 三次，分别采用 `general`、`fixed` 和 `scientific` 格式。哪种格式呈现给用户最精确的表示形式？说明原因。

程序员可以使用操纵符 `setprecision()` 来设置精度，例如：

```
cout << 1234.56789 << '\t'
    << fixed << 1234.56789 << '\t'
    << scientific << 1234.56789 << '\n';
cout << general << setprecision(5)
    << 1234.56789 << '\t'
    << fixed << 1234.56789 << '\t'
    << scientific << 1234.56789 << '\n';
cout << general << setprecision(8)
    << 1234.56789 << '\t'
    << fixed << 1234.56789 << '\t'
    << scientific << 1234.56789 << '\n';
```

会输出(注意舍入)如下结果：

```
1234.57      1234.567890  1.234568e+003
1234.6 1234.56789  1.23457e+003
1234.5679    1234.56789000  1.23456789e+003
```

几种格式的精度分别定义为：

浮点数精度	
general	精度就是数字的个数
scientific	精度为小数点之后数字的个数
fixed	精度为小数点之后数字的个数

一般使用默认的精度为 6 的 `general` 格式即可，除非有特殊情况——一个常见的原因是“我们需要更为精确的输出”。

11.2.5 域

使用科学记数法(`scientific`)和定点(`fixed`)格式，程序员可以精确控制一个值输出所占用的宽度。显然，这对于打印表格这类应用来说很有用。整数输出也有类似的机制，称为域(*field*)。你可以使用“设置域宽度”操纵符 `setw()` 精确指定一个整数或一个字符串输出占用多少个位置。例如：

```
cout << 123456 // no field used
    << '|' << setw(4) << 123456 << '|' // 123456 doesn't fit in a 4-char field
    << setw(8) << 123456 << '|' // set field width to 8
    << 123456 << "\n"; // field sizes don't stick
```

会输出如下结果：

```
123456|123456| 123456|123456|
```

首先注意第三个 123456 之前的两个空格，这就是我们所期望的效果——一个 6 位数字的数占用一个 8 个字符的域。但是，当你指定一个 4 个字符的域时，123456 不会被截取来适应域宽。为什么不进行截取呢？|1234| 或 |3456| 对于宽度为 4 的域来说都是适合的，但它们完全改变了要打印的值，而且没有给用户任何警告信息。`ostream` 是不会这样做的，相反，它会打破输出格式。坏的格式几乎永远比“坏的输出数据”要更好些。而且在使用域最多的应用中(例如打印表格)，“溢出”问题是很容易注意到的，因此能得到修正。

域也可用于浮点数和字符串，例如：

```
cout << 12345 << '|' << setw(4) << 12345 << '|'
    << setw(8) << 12345 << '|' << 12345 << "\n";
cout << 1234.5 << '|' << setw(4) << 1234.5 << '|'
    << setw(8) << 1234.5 << '|' << 1234.5 << "\n";
cout << "asdfg" << '|' << setw(4) << "asdfg" << '|'
    << setw(8) << "asdfg" << '|' << "asdfg" << "\n";
```

会输出如下结果：

```
12345|12345| 12345|12345|
1234.5|1234.5| 1234.5|1234.5|
asdfg|asdfg| asdfg|asdfg|
```

注意，域宽度不是持久的。在三个例子中，第一个和最后一个值的输出方式是“它需要多少位置就占用多少位置”。换句话说，除非你在语句中输出操作之前即时设置域宽，否则不会有域的限制。

试一试 编写程序，创建一个简单的表格，包括你自己和至少五位朋友的姓、名、电话号码、email 地址等信息。试验不同的域宽，直至表格输出形式达到你满意的程度为止。

11.3 文件打开和定位

从 C++ 程序的角度看，文件是操作系统提供的一个抽象。如 10.3 节所述，一个文件就是一个从 0 开始编号的简单的字节序列如下图所示。

问题是我们如何访问这些字节。如果使用 `iostream`，访问方式很大程度上在我们打开文件，将其与一个流相关联时就确定了。流的属性决定了文件打开后我们可以对它执行哪些操作，以及这些操作的意义。最简单的一个例子是，如果我们打开文件关联至一个 `istream`，我们可以从文件读取数据，而使用 `ostream` 打开文件的话，我们可以向文件写入数据。

11.3.1 文件打开模式

C++ 提供了多种文件打开模式。默认情况下，用 `ifstream` 打开的文件用于读，用 `ofstream` 打开的文件用于写，这满足了大多数一般需求。但是，你还可以选择其他方式：

文件流打开模式	
<code>ios_base::app</code>	追加模式(即添加在文件末尾)
<code>ios_base::ate</code>	“末端”模式(打开文件并定位到文件尾)
<code>ios_base::binary</code>	二进制模式——注意系统特有的行为
<code>ios_base::in</code>	用于读模式
<code>ios_base::out</code>	用于写模式
<code>ios_base::trunc</code>	将文件长度截为 0

在创建流对象时，可在文件名之后指定文件模式，例如：

```
ofstream of1(name1); // defaults to ios_base::out
ifstream if1(name2); // defaults to ios_base::in

ofstream ofs(name, ios_base::app); // ofstreams are by default out
fstream fs("myfile", ios_base::in|ios_base::out); // both in and out
```

在后一个例子中的“|”是“位或”运算符(参见附录 A.5.5)，可用于组合多个模式。`app` 模式常用于写日志文件，因为你总是将新的日志追加到文件末尾。

在每个例子中，打开文件的确切效果依赖于操作系统，而且如果操作系统不能使用某种特定的模式打开文件的话，流可能会进入非 `good()` 状态：

```
if (!fs) // oops: we couldn't open that file that way
```

以读模式打开一个文件，最常见的失败原因是文件不存在(至少文件名不是我们所指定的那样)：

```
ifstream ifs("redungs");
if (!ifs) // error: can't open "readings" for reading
```

在本例中，我们猜测是拼写错误导致了文件打开失败。

注意，如果以写模式打开一个文件，而文件不存在的话，通常操作系统会创建一个新文件，但如果以读模式打开一个不存在的文件，就不会创建新文件(这实际上是很幸运的)。

```
ofstream ofs("no-such-file"); // create new file called no-such-file
ifstream ifs("no-file-of-this-name"); // error: ifs will be not be good()
```

11.3.2 二进制文件

在内存中，我们可以将整数 123 表示为一个整型值或者一个字符串值，如下所示：

```
int n = 123;
string s = "123";
```

在第一条语句中，123 保存为一个(二进制)数，与所有其他整型值占用相同大小的内存空间(在 PC 机上是 4 字节，32 位)。即便我们处理数值 12345，仍然占用 4 个字节。在第二条语句中，123 保存为一个三个字符的字符串。如果我们处理的是 12345，则保存为字符串“12345”需占用 5 个字符(还需要加上管理字符串所需的固定开销)。这种差异如下图所示(可以看到，使用普通的十进制和字符表示方式，不如使用在计算机内部采用的二进制表示方式)，如右图所示。

123 存为字符：

1	2	3	?	?	?	?	?
---	---	---	---	---	---	---	---

12345 存为字符：

1	2	3	4	5	?	?	?
---	---	---	---	---	---	---	---

123 存为二进制：

123			
-----	--	--	--

12345 存为二进制：

12345			
-------	--	--	--

当我们使用字符表示方式时，必须使用特定字符表示数值的结束，就像我们在纸上书写数值一样：123456 是一个数，而 123 456 是两个数。在纸上书写，我们使用空格来表示数值的结束。在计算机内存中，我们也可以这么做，如下图所示。

固定长度的二进制表示方式(比如 int 型值)和变长的字符串表示方式(比如 string 类型)之间的差别在文件中也有体现。默认情况下，iostream 使用字符表示方式。也就是说，istream 从文件读取字符序列，并将其转换为所需类型的对象。而 ostream 将指定类型的对象转换为字符序列，然后写入文件。但是，我们可以令 istream 和 ostream 将对象在内存中对应的字节序列简单地复制到文件。这称为二进制(binary) I/O，通过在打开文件时指定 ios_base::binary 模式来实现。下面的例子展示了如何读写二进制整数文件，涉及“二进制”处理的代码将在后面给出详细解释：

```
int main()
{
    // open an istream for binary input from a file:
    cout << "Please enter input file name\n";
    string name;
    cin >> name;
    ifstream ifs(name.c_str(), ios_base::binary); // note: stream mode
    // "binary" tells the stream not to try anything clever with the bytes
    if (!ifs) error("can't open input file ", name);

    // open an ostream for binary output to a file:
    cout << "Please enter output file name\n";
    cin >> name;
    ofstream ofs(name.c_str(), ios_base::binary); // note: stream mode
    // "binary" tells the stream not to try anything clever with the bytes
    if (!ofs) error("can't open output file ", name);
```

```

vector<int> v;

// read from binary file:
int i;
while (ifs.read(as_bytes(i), sizeof(int))) // note: reading bytes
    v.push_back(i);

// ... do something with v ...

// write to binary file:
for(int i=0; i<v.size(); ++i)
    ofs.write(as_bytes(v[i]), sizeof(int)); // note: writing bytes
return 0;
}

```

打开二进制文件是通过指定 `ios_base::binary` 模式实现的：

```

ifstream ifs(name.c_str(), ios_base::binary);

ofstream ofs(name.c_str(), ios_base::binary);

```

在本例中，我们使用相对来讲较为复杂，但也更为紧凑的二进制表示方式。当我们从面向字符的 I/O 转向二进制 I/O 时，要放弃常用的 `>>` 和 `<<` 操作符。这两个操作符按默认约定将值转换为字符序列（例如，字符串“asdf”转换为字符 a、s、d、f，整数 123 转换为字符 1、2、3）。如果我们需要的就是这些，那么也就不必使用二进制了，因为默认模式已经够用了。只有在默认模式不能满足需求时，我们才需要使用二进制文件。我们使用二进制模式，就是告知流，不要试图对字节序列做任何“聪明”的处理。

我如何处理 `int` 型值才是“聪明的”？答案是显然的——用 4 个字节存储 4 字节宽的 `int` 型值，也就是说，我们可以查看 `int` 型值在内存中的表示方式（4 个字节的序列）并直接将这些字节传输到文件。随后，我们就可以用同样的方式读回这些字节重组出 `int` 值：

```

ifs.read(as_bytes(i), sizeof(int)) // note: reading bytes
ofs.write(as_bytes(v[i]), sizeof(int)) // note: writing bytes

```

`ostream` 的 `write()` 函数和 `istream` 的 `read()` 函数都接收两个参数：地址（这里用函数 `as_bytes()` 获取）和字节（字符）数量（这里我们用运算符 `sizeof` 获得）。对于我们要读/写的值，地址参数指向保存它的内存区域的第一个字节。例如，如果我们处理一个 `int` 型值 1234，其内存区域中保存这么 4 个字节的值：00、00、04、d2（十六进制表示），如下图所示：



函数 `as_byte()` 可以用来获取对象存储区域的第一个字节。它可以定义如下（其中用到了我们还未介绍的语言特性，参见 17.8 节和 19.3 节）：

```

template<class T>
char* as_bytes(T& i) // treat a T as a sequence of bytes
{
    void* addr = &i; // get the address of the first byte
                    // of memory used to store the object
    return static_cast<char*>(addr); // treat that memory as bytes
}

```

使用 `static_cast` 的（不安全）类型转换是必需的，我们需要用它来获得一个变量的“原始字节表示”。地址的概念我们会在第 17 章和第 18 章进行详细介绍。在这里，我们只展示如何将内存中

的对象按字节序列的方式处理，供 `read()` 和 `write()` 使用。

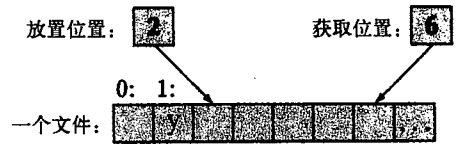
这种二进制 I/O 方式有些困难、有些复杂，而且容易出错。但是作为程序员，我们不是总有选择文件格式的自由。因此，偶尔我们必须使用二进制文件，只是因为我们要读写的文件的制作者选择了二进制格式。另外，也有可能使用非字符表示方式是一种更好的选择。典型的例子是图像和声音文件，它们都没有适合的字符表示方式：一幅照片或者一段音乐本质上就是一个比特包。

`iostream` 库默认的字符 I/O 是可移植的、人类可读的，而且很好地被类型系统所支持。如果条件允许，请尽量使用字符 I/O，除非不得已，否则不要使用二进制 I/O。

11.3.3 在文件中定位

只要有可能，请尽量使用从头至尾的文件读写方式。这是最简单也最不容易出错的方式。很多时候，当你觉得必须对文件进行修改时，最好的方法是创建一个新的文件。

但是，如果你必须使用文件定位功能的话，C++ 也支持在文件中定位到指定位置以进行读写。基本上，每个以读方式打开的文件，都有一个“读/获取位置”，而每个以写方式打开的文件，都有一个“写/放置位置”，如右图所示。



使用方法如下：

```
fstream fs(name.c_str()); // open for input and output
if (!fs) error("can't open ", name);

fs.seekg(5); // move reading position ("g" for "get") to 5 (the 6th character)
char ch;
fs >> ch; // read and increment reading position
cout << "character[5] is " << ch << " ('<int(ch) << "\n";

fs.seekp(1); // move writing position ("p" for "put") to 1
fs << 'y'; // write and increment writing position
```

请小心：这段代码中在文件定位之前进行了运行时错误检测，这是必要的。另外要特别注意的是，如果你试图用 `seekg()` 或 `seekp()` 定位到文件尾之后，结果如何是未定义的，不同操作系统会表现出不同的行为。

11.4 字符串流

你可以将一个 `string` 对象作为 `istream` 的源或 `ostream` 的目标。从一个字符串读取内容的 `istream` 对象称为 `istringstream`，保存字符并将其写入字符串的 `ostream` 对象称为 `ostringstream`。例如，从字符串提取数值时 `istringstream` 就很有用：

```
double str_to_double(string s)
// if possible, convert characters in s to floating-point value
{
    istringstream is(s); // make a stream so that we can read from s
    double d;
    is >> d;
    if (!is) error("double format error: ", s);
    return d;
}

double d1 = str_to_double("12.4"); // testing
double d2 = str_to_double("1.34e-3");
double d3 = str_to_double("twelve point three"); // will call error()
```


如果你试图从一个 `stringstream` 流的字符串尾之后读取字符, 流会进入 `eof()` 状态。这意味着你可以将“标准输入循环”应用于 `stringstream` 流, 实际上一个字符串流就是一个真正的 `istream`。

相反, 对于要求一个简单字符串参数的系统, 如 GUI 系统(参见 16.5 节), `ostringstream` 可用于格式化输出来生成单一字符串。例如:

```
void my_code(string label, Temperature temp)
{
    // ...
    ostringstream os;    // stream for composing a message
    os << setw(8) << label << ": "
        << fixed << setprecision(5) << temp.temp << temp.unit;
    someobject.display(Point(100,100), os.str().c_str());
    // ...
}
```

`stringstream` 的成员函数 `str()` 返回由输出到 `ostringstream` 对象的内容构成的字符串。`c_str()` 是 `string` 的成员函数, 它返回很多系统接口所要求的 C 风格字符串。

`stringstream` 通常用于将真实 I/O 和数据处理分离。例如, `str_to_double()` 的 `string` 参数通常来自一个文件(比如, 一个 Web 日志)或键盘。类似地, 我们在 `my_code()` 中生成的消息最终会输出到屏幕的某个区域。再如, 在 11.7 节中, 我们使用 `stringstream` 来过滤输入中不希望出现的字符。因此, `stringstream` 可以看做一种剪裁 I/O 以适应特殊需求和偏好的机制。

`ostringstream` 的一个简单应用是连接字符串:

```
int seq_no = get_next_number();    // get the number of a log file
ostringstream name;
name << "myfile" << seq_no;        // e.g., myfile17
ofstream logfile(name.str().c_str()); // e.g., open myfile17
```

通常情况下, 我们用一个 `string` 来初始化 `istringstream`, 然后用输入操作从该字符串中读取字符。相反, 我们通常用一个空字符串初始化 `ostringstream`, 然后用输出操作向其中填入字符。有一种更为直接的方法来访问 `stringstream` 中的字符: `ss.str()` 返回 `ss` 的字符串的一个拷贝, 而 `ss.str(s)` 则将 `ss` 的字符串设置为 `s` 的一个拷贝。这种方法某些情况下很有用, 11.7 节展示了一个使用 `ss.str(s)` 的例子, 体现了它的用处。

11.5 面向行的输入

>> 操作符按照对象类型的标准格式读取输入, 保存到对象中。例如, 当读取一个 `int` 型对象时, >> 会一直读取数字, 直至遇到一个非数字的字符为止; 当读取一个 `string` 时, >> 会一直读取字符, 直至遇到空白符为止。标准库 `istream` 库也提供了读取单个字符和整行内容的功能。考虑下面的代码, 它显然没有达到我们预想的效果:

```
string name;
cin >> name;    // input: Dennis Ritchie
cout << name << '\n'; // output: Dennis
```

如果希望一次读取整行内容, 随后再决定如何从中格式化输入数据, 我们应该怎么做呢? 可以使用函数 `getline()`, 例如:

```
string name;
getline(cin, name);    // input: Dennis Ritchie
cout << name << '\n'; // output: Dennis Ritchie
```

现在我们已经获得整行内容了。我们为什么要这么做呢? 一个很好的答案是: “因为我们需要做一些 >> 做不了的事。”通常, 一个不好的答案是: “因为用户输入的就是一整行。”如果你能想到的

使用 `getline()` 的最佳理由只是第二个答案的话, 还是继续使用 `>>` 吧, 因为一旦读入了一整行, 通常情况下你就必须自己来分析输入内容, 例如:

```
string first_name;
string second_name;
stringstream ss(name);
ss>>first_name;           // input Dennis
ss>>second_name;          // input Ritchie
```

显然, 直接用 `>>` 将读入字符串, 存入 `first_name` 和 `second_name` 更简单些。

使用整行输入的一个常见原因是, 默认的空白符不符合我们的要求。有时, 我们可能需要将换行符同其他空白符区别对待。例如, 与一个电脑游戏的文本交互中, 可能将一行作为一句话, 而不使用习惯的空白符。

```
go left until you see a picture on the wall to your right
remove the picture and open the door behind it. take the bag from there
```

对于此例, 我们可以先读入一行, 然后从中提取单个单词。

```
string command;
getline(cin, command);    // read the line

stringstream ss(command);
vector<string> words;
string s;
while (ss>>s) words.push_back(s);    // extract the individual words
```

但是, 只要我们有其他解决方法, 还是尽量使用习惯的空白符(使用 `>>` 读取输入)而不是换行(使用整行输入)。

11.6 字符分类

通常, 我们按习惯格式读入整数、浮点数、单词等。但是, 我们可以(有时是必须)在更低的抽象层上读入单个字符。这样做在编程上需要做更多工作, 但我们能对输入有完全的控制。回顾一下表达式单词分析问题(参见 7.8.2 节), 假如我们希望将 $1 + 4 * x \leq y / z * 5$ 分解为 11 个单词:

$$1 + 4 * x \leq y / z * 5$$

我们可以用 `>>` 读入数值, 但试图以字符串类型读入标识符时, 就会导致 $x \leq y$ 被作为一个字符串读入(因为 `<` 和 `=` 不是空白符), $z *$ 也是如此(因为 `*` 也不是一个空白符)。我们可以写出如下代码实现正确的单词分解:

```
char ch;
while (cin.get(ch)) {
    if (isspace(ch)) {    // if ch is whitespace
        // do nothing (i.e., skip whitespace)
    }
    if (isdigit(ch)) {
        // read a number
    }
    else if (isalpha(ch)) {
        // read an identifier
    }
    else {
        // deal with operators
    }
}
```

函数 `istream::get()` 读入单个字符, 赋予它的参数, 这样不跳过空白符。与 `>>` 类似, `get()` 返回其

istream 对象的引用，我们检测其状态。

当我们采用逐个字符读取方式时，通常需要对字符进行分类：这个字符是数字吗？这个字符是大写字母吗？等等。下面是实现字符分类的标准库函数：

字符分类	
isspace(c)	c 是空白符吗(' ','\t','\n'等)
isalpha(c)	c 是字母吗('a'..'z','A'..'Z')(注意：不包括 '_')
isdigit(c)	c 是十进制数字吗('0'..'9')
isxdigit(c)	c 是十六进制数字吗(十进制数字或 'a'..'f'或 'A'..'F')
isupper(c)	c 是大写字母吗
islower(c)	c 是小写字母吗
isalnum(c)	c 是字母或十进制数字吗
isctrl(c)	c 是控制字符吗(ASCII 码 0..31 和 127)
ispunct(c)	c 是标点(除字母、数字、空白符或不可见控制字符之外的字符)吗
isprint(c)	c 是可打印字符吗(ASCII 字符 ' '.. '~')
isgraph(c)	c 是字母、十进制数字或标点吗(注意：不包括空白符)

注意，多个字符分类可以用“或”运算符(|)进行组合。例如，isalnum(c)意味着 isalpha(c) | isdigit(c)，也就是说，“c 是一个字母或一个数字吗”？

另外，标准库还提供了另外两个有用的函数，用来转换大小写：

字符大小写	
toupper(c)	c 或 c 对应的大写字母
tolower(c)	c 或 c 对应的小写字母

如果你想忽略大小写，这两个函数很有用。例如，用户输入 Right、right 和 rigHT 很可能是想表示相同的意思(rigHT 很可能是不小心误按了 Caps Lock 键)。对这些字符串的每个字符都应用 tolower()后，所有字符串都变为了 right。我们可以为字符串定义 tolower 函数：

```
void tolower(string& s)    // put s into lower case
{
    for (int i=0; i<s.length(); ++i) s[i] = tolower(s[i]);
}
```

参数传递上我们采用了传引用方式(参见 8.5.5 节)，以便函数能真正改变实参字符串。如果我们希望保持原字符串的内容不变，可以编写一个函数，创建原字符串的一个小写拷贝。在处理这类问题时，使用 tolower()比使用 toupper()更好些。因为对于某些自然语言(如德语)，并不是所有小写字母都有对应的大写字母，因此前者能获得更好的效果。

11.7 使用非标准分隔符

本节提供一个接近实际的例子，它使用 istream 库解决一个真实问题。当我们读入字符串时，是以空白符作为默认分隔符的。不幸的是，istream 没有提供自定义分隔符的功能，也不能直接改变 >> 读入字符串的方式。于是，如果我们需要定义其他分隔符，应该怎么做呢？回顾 4.6.3 节中的例子，在那个例子中，我们读入“单词”并进行比较。那些单词都是以空白符分隔的，因此，如果我们输入

As planned, the guests arrived; then,

我们会得到这些“单词”：

```
As
planned,
the
guests
arrived;
then,
```

对于“planned,”和“arrived;”，我们在字典中是找不到这些字符串的，它们并不是单词。它们实际上是由单词加上毫无关系的、分散注意力的标点字符构成的。而在大多数场合下，我们是应该将标点与空白符等同对待的。那么该如何去掉这些标点呢？我们可以逐个处理字符，将标点字符删除或者转换为空白符，随后再从“清理干净”的输入中读取数据：

```
string line;
getline(cin,line);           // read into line
for (int i=0; i<line.size(); ++i) // replace each punctuation character
                                // by a space
    switch(line[i]) {
        case ';': case '.': case ',': case '?': case '!':
            line[i] = ' ';
    }

stringstream ss(line);       // make an istream ss reading from line
vector<string> vs;
string word;
while (ss>>word)              // read words without punctuation characters
    vs.push_back(word);
```

同样是前面给出的输入，这段代码会得到如下单词：

```
As
planned
the
guests
arrived
then
```

不幸的是，这段代码有些乱，而且是专用而非通用的。如果对另外一个问题，标点集发生变化，我们又该怎么办呢？下面我们提出一种更为通用、更为有效的从输入流中删除不需要字符的方法。这种方法应该是怎么样的呢？我们希望使用这一功能的用户程序是什么样的呢？考虑下面的代码：

```
ps.whitespace(";,."); // treat semicolon, colon, comma, and dot as whitespace
string word;
while (ps>>word) vs.push_back(word);
```

我们如何才能定义一个像 `ps` 这样的流呢？基本思想是先从一个普通输入流读入单词，然后使用用户指定的“空白符”来处理输入内容，也就是说，我们并不将“空白符”交给用户，我们只是用它们来分隔单词。例如

```
as.not
```

应该是两个单词

```
as
not
```

我们可以定义一个类来实现上述功能。这个类必须从一个 `istream` 读取数据，而且要有一个 `>>` 操作符，能像 `istream` 一样工作，唯一的差别是我们可以告知它哪些字符作为空白符。简单起见，我们不支持默认空白符（空格、换行等）。也就是说，我们只允许用户指定非标准的空白符。我们也不提供从输入流中完全删除指定字符的功能，只是将它们转换为标准空白符。我们将这个类命名

为 Punct_stream:

```
class Punct_stream {    // like an istream, but the user can add to
                        // the set of whitespace characters

public:
    Punct_stream(istream& is)
        : source(is), sensitive(true) { }

    void whitespace(const string& s)    // make s the whitespace set
        { white = s; }
    void add_white(char c) { white += c; }    // add to the whitespace set
    bool is_whitespace(char c);    // is c in the whitespace set?

    void case_sensitive(bool b) { sensitive = b; }
    bool is_case_sensitive() { return sensitive; }

    Punct_stream& operator>>(string& s);
    operator bool();

private:
    istream& source;        // character source
    istringstream buffer;    // we let buffer do our formatting
    string white;        // characters considered "whitespace"
    bool sensitive;        // is the stream case-sensitive?
};
```

如上面示例代码所示, 基本思想是从 istream 中一次读取一行, 将指定的空白符转换为空格, 然后使用 istringstream 完成格式化。除了处理用户自定义空白符外, 我们还为 Punct_stream 实现了一个相关的功能: 我们可以通过 case_sensitive() 要求它将大小写敏感的输入转换为大小写不敏感的输入。例如, 我们可以要求 Punct_stream 将

Man bites dog!

转换为

man
bites
dog

Punct_stream 的构造函数接受一个 istream 参数作为字符输入源, 并将其命名为 source。构造函数还将流默认设置为大小写敏感的。下面代码可以令 Punct_stream 从 cin 读入字符, 并将分号、冒号和点作为空白符, 并将所有字符转换为小写:

```
Punct_stream ps(cin);    // ps reads from cin
ps.whitespace(";.");    // semicolon, colon, and dot are also whitespace
ps.case_sensitive(false);    // not case-sensitive
```

显然, 最有趣的操作是输入操作符 >>, 这也是目前为止最难于实现的。基本策略是从 istream 读取一整行, 存入一个名为 line 的字符串, 然后将所有自定义空白符转换为空格符 ' '。完成后, 我们将 line 放入名为 buffer 的 istringstream。现在就可以使用识别一般空白符的 >> 从 buffer 中读取数据了。实际代码比上述过程稍微复杂一些, 因为从 buffer 中读取数据直接就可以进行, 但只有在它为空的情况下, 才能向其写入内容。

```
Punct_stream& Punct_stream::operator>>(string& s)
{
    while (!(buffer>>s)) {    // try to read from buffer
        if (buffer.bad() || !source.good()) return *this;
        buffer.clear();

        string line;
        getline(source, line);    // get a line from source
```

```

        // do character replacement as needed:
        for (int i = 0; i < line.size(); ++i)
            if (is_whitespace(line[i]))
                line[i] = ' ';           // to space
            else if (!sensitive)
                line[i] = tolower(line[i]); // to lower case

        buffer.str(line);                // put string into stream
    }
    return *this;
}

```

我们逐行分析一下这段程序。先看一下这行有点不寻常的代码

```
while (!(buffer >> s)) {
```

如果名为 `buffer` 的 `istream` 中存有字符，读操作 `buffer >> s` 就可以进行，`s` 会收到以“空白符”分隔的单词，随后就没有什么可做的了。只要 `buffer` 中有我们可以读取的字符，这个过程就会发生。但是，当 `buffer >> s` 失败时，也就是 `!(buffer >> s)` 为真时，我们必须利用 `source` 中的内容将 `buffer` 重新填满。注意读操作 `buffer >> s` 是在一个循环中，因此当我们尝试重新填充 `buffer` 后，会再次尝试这个读操作，因此有如下代码：

```

while (!(buffer >> s)) {    // try to read from buffer
    if (buffer.bad() || !source.good()) return *this;
    buffer.clear();

    // replenish buffer
}

```

如果 `buffer` 处于 `bad()` 状态，或者 `source` 有问题的话，我们将放弃读取操作。否则，清空 `buffer`，再次尝试。清空 `buffer` 的原因是，只有在读失败的情况下，通常是 `buffer` 遇到 `eof()` 时，我们才进入“重新填充循环”，而这意味着 `buffer` 中没有供我们读取的字符。处理流状态总是很麻烦的，而且常常是微妙错误的根源，需要冗长的调试过程来消除。幸运的是，重填循环的剩余部分很简单：

```

string line;
getline(source, line); // get a line from source

// do character replacement as needed:
for (int i = 0; i < line.size(); ++i)
    if (is_whitespace(line[i]))
        line[i] = ' ';           // to space
    else if (!sensitive)
        line[i] = tolower(line[i]); // to lower case

buffer.str(line);        // put string into stream

```

我们先读入一整行到 `line`，然后检查其中的每个字符，看是否需要改变。`is_whitespace()` 是 `Punct_stream` 的成员函数，我们随后再定义它。`tolower()` 是标准库函数，其功能显然是将字母转换为小写，如将 `A` 转换为 `a`（参见 11.6 节）。

一旦我们正确处理完 `line` 中的内容，就需要将其存入 `istream`。`buffer.str(line)` 完成这一工作，这条语句可以读做“将 `istream` 对象 `buffer` 的字符串值设置为 `line` 的内容”。

注意，使用 `getline()` 从 `source` 读入一行字符后，我们“忘记了”检测它的状态。实际上不必那么做，原因是我们最终会到达位于循环顶部的 `!source.good()` 语句，这时就会进行检测了。

这里我们仍然将流自身的引用——`*this` 作为 `>>` 的返回结果，参见 17.10 节。

测试空白符的部分很容易实现，我们只需将输入字符与所有指定空白符一一进行比较即可：

```
bool Punct_stream::is_whitespace(char c)
{
    for (int i = 0; i < white.size(); ++i) if (c == white[i]) return true;
    return false;
}
```

记住，我们将处理默认空白符（如换行和空格）的工作留给 `istream` 来做了，因此我们无需对这类空白符做特殊处理。

只剩下一个神秘的函数了：

```
Punct_stream::operator bool()
{
    return !(source.fail() || source.bad()) && source.good();
}
```

`istream` 的一种习惯用法是测试 `>>` 的结果，例如：

```
while (ps >> s) { /* ... */ }
```

这意味着我们需要一种方法将 `ps >> s` 的结果当做布尔值来进行检查。但 `ps >> s` 的结果是一个 `Punct_stream`，因此我们需要一种方法将 `Punct_stream` 隐式转换为 `bool` 值。这就是 `Punct_stream` 的 `operator bool()` 所做的事情。`Punct_stream` 的名为 `operator bool()` 的成员函数定义了流到 `bool` 类型的转换。特别地，它在 `Punct_stream` 上的操作成功时返回 `true`。

现在我们可以写程序来测试 `Punct_stream` 类了。

```
int main()
    // given text input, produce a sorted list of all words in that text
    // ignore punctuation and case differences
    // eliminate duplicates from the output
{
    Punct_stream ps(cin);
    ps.whitespace(" ; , . ? ! ( ) { } < > / & $ @ # % ^ * | ~ \" ); // note \" means \" in string
    ps.case_sensitive(false);

    cout << "please enter words\n";
    vector<string> vs;
    string word;
    while (ps >> word) vs.push_back(word); // read words

    sort(vs.begin(), vs.end()); // sort in lexicographical order
    for (int i = 0; i < vs.size(); ++i) // write dictionary
        if (i == 0 || vs[i] != vs[i - 1]) cout << vs[i] << endl;
}
```

这个程序会将输入中出现的单词排序输出。测试语句

```
if (i == 0 || vs[i] != vs[i - 1])
```

会去除重复单词。试一试给这个程序下面的输入内容：

```
There are only two kinds of languages: languages that people complain
about, and languages that people don't use.
```

程序会输出

```
and
are
complain
don't
languages
of
only
people
```

```
that
there
two
use
```

为什么会得到 `don't` 而不是 `dont` 呢？因为我们并没有将单引号指定为空白符。

注意, `Punct_stream` 在很多重要的方面都与 `istream` 相似,但它的确不是 `istream`。例如,我们不能使用 `rdstate()` 来获取流状态, `eof()` 也没有定义,而且我们也不必为针对整数的 `>>` 而烦恼(`Punct_stream` 只是用来处理字符串的)。重要的是,对于一个期望 `istream` 参数的函数,我们不能将 `Punct_stream` 传递给它。我们可以使 `Punct_stream` 真的成为 `istream` 吗?当然可以,但现在我们所拥有的编程经验、设计思想和语言工具还不足以实现这一目标(如果你以后希望回过头再来完成这个目标——当然是很久以后的事情,你还必须从某些专家水平的指南或手册中深入学习流缓冲相关的内容)。

你是否发现 `Punct_stream` 的代码很易读?是否注意到注释很容易理解?你是否认为你自己也能编写这些代码?如果你几天前还是一个真正的初学者,诚实的回答应该是“不!不!不!”甚至是“不!不!绝不可能!!你疯了吗?”我们理解你的想法,但是对于最后一个疑问,回答确实是“不,至少我们认为不可能”。我们给出本例的目的是:

- 展示一个相对实际的问题和解决方案。
- 展示用相对适中的方法能达到什么样的结果。
- 对于一个显然较为简单的问题,给出一个易于使用的解决方案。
- 说明接口和实现之间的差别。

为了成为一名程序员,你需要阅读真实代码,而不仅仅是用于教学的“精致的”解决方案。我们给出这个例子就是出于这个目的。过了几天或几周后,这个程序对你来说就会很容易理解了,你就可以考虑改进它了。

我们可以这样看待这个例子,它就相当于教师在英语初学者课上讲授了一些真正的英语俚语,为课程增加了一些色彩,活跃了学习气氛。

11.8 还有很多未讨论的内容

I/O 相关的细节问题看上去是无穷无尽的,因为它们只受限于人类的创造力和想象力。这就如同我们无法想象自然语言有多复杂一样。英语中的 12.35 按大多数其他欧洲语言的习惯应该表示为 12,35。自然地, C++ 标准库提供了处理这一问题以及其他很多自然语言相关的 I/O 问题的功能。但是,你如何输出中文符号呢?你如何比较两个马拉雅拉姆语字符串呢?这些问题已经有解决方案了,但这些内容远远超出了本书的讨论范围。如果你对此感兴趣,请参考更为专门的或高阶的书籍(如 Langer 的《Standard C++ IOStreams and Locales》和 Stroustrup 的《The C++ Programming Language》),以及标准库和系统的文档。请搜索“本地化”(locale)一词,这个术语通常用于描述处理自然语言差异的程序设计语言特性。

另一个复杂性之源是缓冲机制:标准库 `istream` 依赖于一个称为 `streambuf` 的机制。对于那些高端的应用(无论是从性能角度还是从功能角度),都不可避免地会用到 `streambuf`。如果你觉得需要定义自己的 `istream`,或者需要调整 `istream` 用于新的数据源/目的,参见 Stroustrup 的《The C++ Programming Language》第 21 章或系统文档。

当使用 C++ 时,你也可能会遇到以 `printf()/scanf()` 为代表的 C 语言标准 I/O 函数族。如果你希望了解这部分内容,请参考 27.6 节和附录 B.10.2,或者参考 Kernighan 和 Ritchie 所编写的优

秀的 C 语言教材《The C Programming Language》，以及互联网上数不清的资源。每种程序设计语言都有其自己的 I/O 机制，各不相同，有的很古怪，但大多数都(以不同方式)反映了我们在第 10 章和第 11 章中所介绍的基本思想。

附录 B 总结了 C++ 标准库的 I/O 机制。

图形用户接口(GUI)相关的话题将在第 12 ~ 16 章进行介绍。

简单练习

1. 开始编写一个名为 Test_output.cpp 的程序。声明一个整数 birth_year，并将你的出生年份赋予它。
2. 以十进制、十六进制和八进制格式输出 birth_year。
3. 标出每个输出值所用的基底的名字。
4. 使用制表符将输出定位到适当的列上。
5. 现在输出你的年龄。
6. 现在有什么问题吗？发生什么情况了？将输出固定为十进制。
7. 返回第 2 题，让输出的每个值都显示基底。
8. 尝试读入八进制数、十六进制数，等等：

```
cin >> a >> oct >> b >> hex >> c >> d;
cout << a << '\t' << b << '\t' << c << '\t' << d << '\n';
```

运行程序，输入下面内容

```
1234 1234 1234 1234
```

解释输出结果。

9. 编写程序，分别以 general、fixed 和 scientific 格式输出浮点数 1234567.89。哪种格式呈现给用户最精确的表示？解释为什么。
10. 创建一个简单的表格，包含你和至少 5 位朋友的姓、名、电话号码和电子邮件地址。试验不同的域宽，直至表格的输出满足你的需求为止。

思考题

1. 为什么 I/O 对于程序员来说有些棘手？
2. 符号 << hex 的作用是什么？
3. 十六进制数在计算机科学中的作用是什么？为什么？
4. 为你想实现的几个整数输出格式化选项命名。
5. 什么是操纵符？
6. 十进制数的前缀是什么？八进制呢？十六进制呢？
7. 浮点数值默认输出格式是什么？
8. 什么是域？
9. 解释 setprecision() 和 setw() 的作用。
10. 文件打开模式的目的是什么？
11. 下面哪个操纵符不是持久的：hex、scientific、setprecision、showbase、setw？
12. 字符串 I/O 和二进制 I/O 的差异在哪里？
13. 给出一个例子，使用二进制文件比使用文本文件更好。
14. 给出两个例子，说明 stringstream 的用途。
15. 什么是文件定位？
16. 如果你试图定位到文件尾之后，会出现什么结果？
17. 什么情况下，使用面向行的输入比面向类型的输入更好？
18. isalnum(c) 的功能是什么？

术语

二进制	十六进制	八进制	字符分类	无规律	输出格式
十进制	面向行的输入	有规律	文件定位	操纵符	scientific
fixed	非标准分隔符	setprecision	general	noshowbase	showbase

习题

- 编写程序，读取一个文本文件，将其中的字母都转换为小写，生成一个新文件。
- 编写程序，将文件中的元音都删除(“disemvowels”，只写辅音的输入方式)。例如，将“Once upon a time!”转换为“nc pn tm!”。令人惊奇的是，通常得到的结果还是可读的。请你的朋友测试这个程序。
- 编写一个名为 multi_input.cpp 的程序，提示用户输入几个整数，可以使用不同的数制，对八进制和十六进制分别使用 0 和 0x 前缀进行输入。程序能正确解释这些数值，并将它们转换为十进制格式。随后输出这些数值，如下图所示，列对齐：

0x4	hexadecimal	converts to	67	decimal
0123	octal	converts to	83	decimal
65	decimal	converts to	65	decimal
- 编写程序，读入一些字符串，对每个字符串，输出其中每个字符的分类，字符分类方式和分类函数如 11.6 节所述。注意，一个字符可能属于多个类别(如 x 既是字母又是字母数字)。
- 编写程序，将输入中的标点转换为空白符。例如，“- don't use the as-if rule.”转换为“dont use the asif rule”。
- 修改习题 5 中的程序，将 don't 转换为 do not, can't 转换为 can not, 等等，在单词内的标点保持不变(于是上题中的输入会转换为“do not use the as-if rule”)。
- 编写程序，利用上题的程序生成字典(替代 11.7 节中的方法)。对一个多页的文本文件运行程序，观察结果是否还有改进的余地。
- 将 11.3.2 节中的二进制 I/O 程序一分为二：一个程序将原始文本文件转换为二进制，另一个程序读入二进制文件，将其转换为文本格式。测试这两个程序：对一个文本文件进行这两个步骤的转换，将结果与原始文件进行比较。
- 编写函数 `vector<string> split(const string& s)`，将 s 中以空白符分隔的子串存入向量，作为结果返回。
- 编写函数 `vector<string> split(const string& s, const string& w)`，与上题的 split 函数相比，新的 split 函数除了默认的空白符外，还将 w 中的字符当做空白符。
- 编写程序，将一个文本文件中的字符颠倒顺序。例如“asdfghjkl”转换为“lkjhgfdsa”。提示：文件打开模式。
- 编写程序，将一个文件中单词(空白符间隔的字符串)的顺序颠倒过来。例如，将 Norwegian Blue parrot 转换为 parrot Blue Norwegian。你可以假定内存空间可以容纳文件中的所有字符串。
- 编写程序，读取一个文本文件，输出文件中包含不同类别字符的个数，字符类别定义如 11.6 节所述。
- 编写程序，读取一个文件，文件格式是以空白符间隔的数值，将这些数值输出到另一个文件，格式采用科学记数法，精度为 8，每行 4 个域，每个域的宽度为 20 个字符。
- 编写程序，读取一个以空白符间隔的数值文件，按升序输出这些数值，每行一个值。每个数值只输出一次，如果一个数值在原文件中出现多次，同时输出它出现的次数。例如，若原文件为“7 5 5 7 3 117 5”，则输出：

```

3
5   3
7   2
117

```

附言

输入/输出是很难处理的，因为我们人类的喜好和习惯并不遵从简单的、易于阐述的原则和直接的数学法则。作为程序员，我们几乎不可能命令用户偏离他自己的习惯和偏好；即便我们可以，最好也不要过于自大，以至于认为我们可以提供一种简单方式，来替代人类千百年来形成的习惯。因此，我们必须预计到输入/输出中所面临的一定程度上的混乱，并接受它、适应它。与此同时，还要尽力保持我们程序的简洁——但不要过度简单化。

第 12 章 一个显示模型

“直到 20 世纪 30 年代，世界才变成彩色的。”

——Calvin's dad

本章描述了一个显示模型(GUI 的输出部分)，并给出了使用方法和一些基本概念，如屏幕坐标、线和颜色等。Shape 是内存中的一个对象，我们可以将其显示在屏幕上并进行适当的操作。Line、Lines、Polygon、Axis 和 Text 都是 Shape 的实例。后面两章将进一步探讨这些类，第 13 章侧重类的实现，第 14 章侧重设计问题。

12.1 为什么要使用图形用户界面

我们为什么用四章篇幅介绍图形并用一章介绍 GUI(图形用户接口)呢？毕竟本书是一本程序设计书籍，而非图形学书籍。实际上有很多非常有趣的软件设计的话题，我们无法一一讨论，在这里只是介绍与图形有关的内容。那么，为什么要选择图形这个话题呢？从本质上讲，图形学是一个学科，在其学习过程中，我们可以对软件设计、程序设计及其语言特性等重要的领域进行深入的探讨。

- 图形很有用。虽然使用 GUI，更多工作仍在于程序设计而非图形，软件设计仍比编写代码更重要。但是，图形在很多领域都非常重要，例如，对于科学计算、数据分析或者任何一个量化问题，没有数据图形化功能是不可想象的。第 15 章给出了一个简单通用的数据图形化工具。
- 图形很有趣。在多数计算领域中，一段代码的效果是很难立刻呈现出来的，即便最后代码没有 bug 了，也无法立刻看出来。这时，即使图形没有用，我们也会倾向于使用图形界面来获得即时效果。
- 图形提供了许多有趣的代码。通过阅读大量程序代码，可以找到一种对代码好坏的判断标准，就像写好英文文章必须要阅读大量书籍、文章和高质量的报纸一样。由于程序代码与屏幕显示有某种直接关系，图形代码比复杂程度接近的其他类别的代码更易于阅读。经过本章几分钟的介绍你就能够阅读图形代码，再经过第 13 章的学习就能编写图形代码。
- 图形设计实例非常丰富。实际上，设计和实现一个好的图形和 GUI 库是很困难的。图形领域中有非常丰富的具体的、贴近实际的例子，可供学习设计策略和设计技术。通过相对较少的图形和 GUI 代码，就能展示包括类的设计、函数的设计、软件分层(抽象)、库的创建等在内的许多技术。
- 图形有利于解释面向对象程序设计的概念及其语言特征。与流言相反的是，面向对象程序最初并不是为了图形化应用所设计的(参见第 22 章)，但它确实很快就应用到图形领域中，而且图形化应用提供了一些非常易于理解的面向对象设计实例。
- 一些关键的图形学概念非常重要，而且不是那么简单直白的。因此应该在教学中进行讲授，而不是靠自己的主动性(以及耐心)去学习理解。如果我们不展示图形和 GUI 程序的实现方法，你可能会认为它们是不可思议的魔法，这显然偏离了本书的一个基本目标。

12.2 一个显示模型

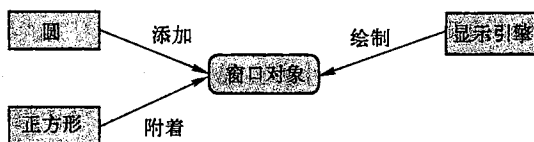
iostream 库是面向字符的输入/输出流,用于处理数值序列或者书籍的文本最为适合。其中, newline 和 tab 控制字符是仅有对图形位置概念的直接的支持。现有的版面设计语言(排版语言、“标注语言”),如 Troff、Tex、Word、HTML、XML(及其配套的图形包),允许在一维字符流中嵌入颜色和二维位置等概念。例如:

```
<hr>
<h2>
Organization
</h2>
This list is organized in three parts:
<ul>
    <li><b>Proposals</b>, numbered EPddd, ...</li>
    <li><b>Issues</b>, numbered EIddd, ...</li>
    <li><b>Suggestions</b>, numbered ESddd, ...</li>
</ul>
<p>We try to ...
</p>
```

这段 HTML 代码指定了一个文档头(<h2>...</h2>),一个包含若干列表项(...)的列表(...)和一个段落(<p>)。注意,我们省略了很多无关的代码。这类语言的关键点是,你可以在普通文本中表示版面的概念,但代码与屏幕上的显示内容之间不是直接关联的,而是由解释这些“标注”命令的程序来控制屏幕上的显示内容。这种技术极为简单,又极为有效(现在你所阅读的所有文档、网页等基本都是这样生成的),但也有其缺点。

本章和之后 4 章介绍另外一种技术:一种图形及图形用户界面直接对应屏幕显示的思想。其基本概念先天就都是图形化的(而且都是二维的,适应计算机屏幕的矩形区域),这些基本概念包括坐标、线、矩形和圆等。从编程的角度看,其目的是建立内存中的对象和屏幕图像的直接对应关系。

其基本模型如下:我们利用图形系统提供的基本对象(如线)组合出更复杂的对象;然后将这些对象添加到一个表示物理屏幕的窗口对象中;最后,用一个程序将我们添加到窗口上的对象显示在屏幕上,我们可以将这个程序看做屏幕显示本身,或者是一个“显示引擎”,或者是“我们的图形库”,或者是“GUI 库”,甚至(幽默地)将其看做“在屏幕背后进行画图工作的小矮人”,然后在屏幕上画出我们要添加到窗口的对象:



“显示引擎”负责在屏幕上绘制线,将文本串放置在屏幕上,为屏幕区域着色,等等。简单起见,我们将使用“我们的 GUI 库”甚至“系统”来表示显示引擎,虽然它的功能不只是绘制对象。与我们的代码调用 GUI 库实现大部分图形功能一样,GUI 库将它的很多工作交由操作系统来完成。

12.3 第一个例子

我们的目标是定义一些类,能够用来创建可以在屏幕上显示的对象。例如,我们希望绘制一个由一系列相连的线构成的图形,下面程序给出了一个非常简单的版本:

```

#include "Simple_window.h" // get access to our window library
#include "Graph.h"         // get access to our graphics library facilities

int main()
{
    using namespace Graph_lib; // our graphics facilities are in Graph_lib

    Point tl(100,100);        // to become top left corner of window

    Simple_window win(tl,600,400,"Canvas"); // make a simple window

    Polygon poly;              // make a shape (a polygon)

    poly.add(Point(300,200)); // add a point
    poly.add(Point(350,100)); // add another point
    poly.add(Point(400,200)); // add a third point

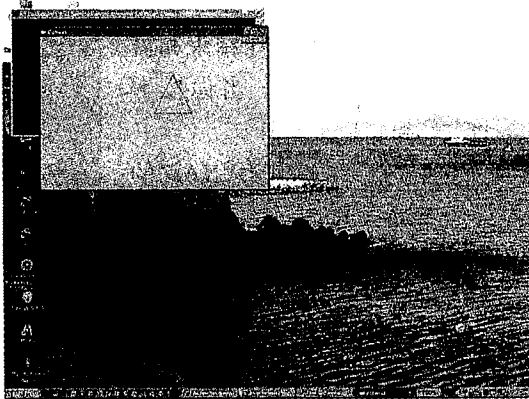
    poly.set_color(Color::red); // adjust properties of poly

    win.attach (poly);         // connect poly to the window

    win.wait_for_button();     // give control to the display engine
}

```

运行该程序，屏幕显示如下：



我们来逐行分析这个程序，看看它做了什么。它首先包含图形接口库的头文件：

```

#include "Simple_window.h" // get access to our window library
#include "Graph.h"         // get access to our graphics library facilities

```

接着，在 `main()` 函数的开始处告知编译器在 `Graph_lib` 中查找图形工具：

```
using namespace Graph_lib; // our graphics facilities are in Graph_lib
```

然后，定义一个点作为窗口的左上角：

```
Point tl(100,100); // to become top left corner of window
```

接下来在屏幕上创建一个窗口：

```
Simple_window win(tl,600,400,"Canvas"); // make a simple window
```

我们使用 `Graph_lib` 接口库中一个名为 `Simple_window` 的类表示窗口，此处定义了一个名为 `win` 的 `Simple_window` 对象，并使用初始化列表中的值将窗口的左上角设置为 `tl`，宽度和高度分别设置为 600 和 400 像素。我们随后会介绍更多细节，但此处的关键点就是通过给定宽度和高度来定义一个矩形。字符串 `Canvas` 用于标识该窗口，你可以在窗口框左上角的位置看到 `Canvas` 字样。

接下来，我们在窗口中放置一个对象：

```

Polygon poly;           // make a shape (a polygon)

poly.add(Point(300,200)); // add a point
poly.add(Point(350,100)); // add another point
poly.add(Point(400,200)); // add a third point

```

我们定义了一个多边形对象 `poly`，并向其添加顶点。在我们的图形库中，一个 `Polygon` 对象开始为空，可以向其中添加任意多个顶点。由于我们添加了三个顶点，因此得到了一个三角形。一个点是一个值对，给出了点在窗口内的 `x` 和 `y`（水平和垂直）坐标。

纯粹是为了展示图形库的功能，我们接下来将多边形的边染为红色：

```
poly.set_color(Color::red); // adjust properties of poly
```

最后，我们将 `poly` 添加到窗口 `win`：

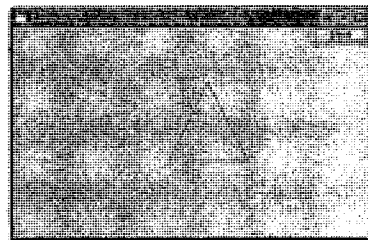
```
win.attach(poly); // connect poly to the window
```

如果程序执行得不是那么快，你会注意到，到现在为止，屏幕上没有任何显示，是的，什么都没有。我们创建了一个窗口（`Simple_window` 类的一个对象），创建了一个多边形（名为 `poly`）并将其染为红色（`Color::red`），最后将其添加到窗口 `win`，但我们还没有要求在屏幕上显示此窗口。显示操作由程序的最后一行代码来完成：

```
win.wait_for_button(); // give control to the display engine
```

为了让 GUI 系统在屏幕上显示一个对象，你必须将控制权交给“系统”。`wait_for_button()` 就是完成这个功能，另外，它还等待用户按下（点击）窗口中的“Next”按钮，以便执行下面的程序。这样，在程序结束和窗口消失之前，你就有机会看到窗口中的内容。当你按下“Next”按钮后，程序会结束，将关闭窗口。

单独地看这个窗口，效果如右图所示。标记为“Next”的按钮是从哪里来的？实际上它是 `Simple_window` 类内置的。在第 16 章中，我们将会从 `Simple_window` 类过渡到“普通”的 `Window` 类，它不包含任何可能造成混淆的内置功能。那时，我们还会介绍如何编写代码来控制与窗口的交互。



在接下来的三章里，当希望逐阶段（一帧一帧）地显示信息时，我们将简单地使用“Next”按钮来实现显示画面的转换。

对于操作系统为每个窗口添加窗口框（frame），你应该非常熟悉了，但可能没有特别留意过。不过，本章和后面章节中的图片都是在微软 Windows 系统下生成的，因此你“免费”得到了窗口框右上角的三个常用按钮。这些按钮是很有用的：如果你的程序变得杂乱无章（在调试过程中确实有可能出现这种情况），可以点击 `x` 按钮结束程序。当你在其他系统上运行程序时，根据系统惯例的不同，添加的窗口框也可能有所不同。在上面的示例程序中，我们对窗口框所做的仅仅是设置了一个标签“Canvas”。

12.4 使用 GUI 库

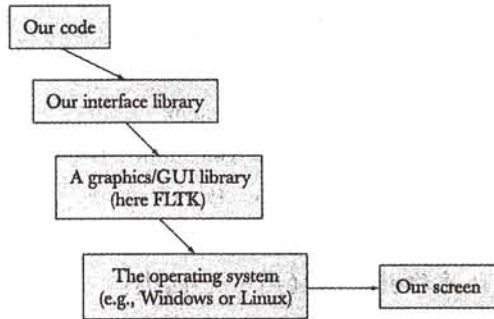
在本书中，我们不直接采用操作系统的图形和 GUI 工具，否则会将程序限制在一种特定的操作系统上，而且需要处理很多复杂的细节问题。与处理文本 I/O 一样，我们将使用一个函数库来消除操作系统间的差异、I/O 设备的变化等问题，并简化程序代码。不幸的是，C++ 并没有提供一个像标准流 I/O 库一样的标准 GUI 库，于是我们从很多可用的 GUI 库选择了一个。为了不局限

于这种 GUI 库, 并且避免一开始就接触其复杂功能, 我们只使用一组在任何 GUI 库中都只需几百行程序就能实现的接口类。

我们使用的(目前还只是间接使用)GUI 工具包名为 FLTK(Fast Light Tool Kit, 读做“full tick”), 具体请参考 www.fltk.org。因此, 我们的代码可以移植到任何使用 FLTK 的平台——包括 Windows、UNIX、Mac、Linux 等。而且我们的接口类也可以使用其他的图形工具包重新实现, 因此基于它的代码的移植性实际上还要更好一些。

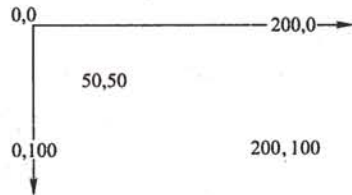
接口类实现的编程模型比通常的工具包提供的更简单。例如, 我们整个图形和 GUI 接口库的 C++ 代码大约为 600 行, 而最简单的 FLTK 文档也达 370 页。你可以从 www.fltk.org 下载, 但我们并不推荐你阅读, 目前还不需要那些细节。第 12 ~ 16 章给出的概念可用于任何一个流行的 GUI 工具包, 当然我们也会解释接口类是如何映射到 FLTK 的, 以便在必要的时候能够直接使用其他的工具包。

我们实现的图形工具包的部分结构如右图所示。接口类为二维形状提供了简单、用户可扩展的基本框架, 并支持简单的颜色。为了实现这些功能, 我们给出了基于“回调函数”的 GUI 概念, 这些函数由屏幕上的用户自定义按钮等组件触发(参见第 16 章)。



12.5 坐标系

计算机屏幕是一个由像素组成的矩形区域, 像素是一个可以设置为某种颜色的点。在程序中, 最常见的方式就是将屏幕建模为由像素组成的矩形区域, 每个像素由 x (水平)坐标和 y (垂直)坐标确定。最左端的像素的 x 坐标为 0, 向右逐步递增, 直到最右端的像素为止; 最顶端的像素的 y 坐标为 0, 向下逐步递增, 直到最底端的像素为止。注意, y 坐标是“向下增长”的。这可能有点奇怪, 特别是对数学家而言。但是, 屏幕(窗口)大小各异, 左上角可能是不同屏幕的唯一共同之处了, 因此将其设定为原点。



不同屏幕的像素数可能各不相同, 常见的尺寸有: 1024×768 、 1280×1024 、 1450×1050 和 1600×1200 。

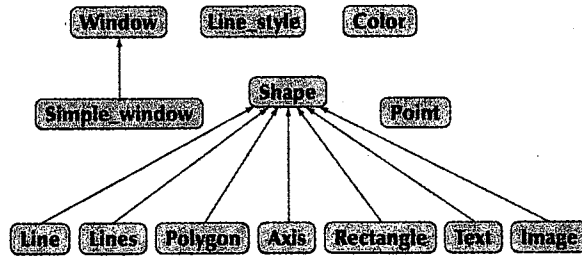
在使用屏幕与计算机进行交互时, 通常从屏幕上划分出特定用途的、由程序控制的矩形区域——窗口。对窗口的操作与屏幕完全一致。基本上, 我们将窗口看做一个小屏幕。例如:

```
Simple_window win(100,600,400,"Canvas");
```

该语句定义了宽度为 600 个像素、高度为 400 个像素的矩形区域, x 坐标从左到右为 0 ~ 599, y 坐标从上到下为 0 ~ 399。能够进行绘制的窗口区域通常称为画布(canvas)。我们指定的大小 600×400 指的就是“内部大小”, 即位于系统提供的窗口框内部的大小, 不包括标题栏、退出按钮等占用的空间。

12.6 形状

我们提供的基本绘图工具包由 12 个类构成:



箭头表示：当需要箭头头部的类时，可以使用尾部的类。例如，当需要一个 Shape 时，我们可以提供一个 Polygon，即 Polygon 是一种 Shape。

我们将从以下类开始进行介绍：

- Simple_window、Window
- Shape、Text、Polygon、Line、Lines、Rectangle、Function 等
- Color、Line_style、Point
- Axis

第 16 章将引入 GUI(用户交互)类：

- Button、In_box、Menu 等

我们可以很容易地添加更多的类(当然取决于你对“容易”的定义)，例如：

- Spline、Grid、Block_chart、Pie_chart 等

然而，定义或描述一个完整的 GUI 框架及其所有功能已经超出了本书的范围。

12.7 使用形状类

本节介绍图形库的一些基本特性：Simple_window、Window、Shape、Text、Polygon、Line、Lines、Rectangle、Color、Line_style、Point、Axis，目的是让你知道这些特性能够实现什么功能，而并非详细理解某个类。第 13 章将会介绍每个类的设计与实现。

下面我们来学习一个简单的程序，我们将逐行解释代码，并说明每一行代码在屏幕上的显示效果。在程序运行时，你会看到当我们向窗口添加形状以及改变已有形状时，屏幕上图像的变化情况。基本上，我们是通过分析程序的执行情况来“动画演示”代码的流程。

12.7.1 图形头文件和主函数

首先，包含定义了图形和 GUI 功能接口的头文件：

```
#include "Window.h" // a plain window
#include "Graph.h"
```

或者

```
#include "Simple_window.h" // if we want that "Next" button
#include "Graph.h"
```

其中，Window.h 包含与窗口有关的特性；Graph.h 包含在窗口上绘制形状(包括文本)的有关特性；这些特性都定义在 Graph_lib 名字空间中。为简化起见，我们使用名字空间指令，使得 Graph_lib 中的名字可以直接在程序中使用。

```
using namespace Graph_lib;
```

按照惯例，main() 函数包含我们要(直接或间接)执行的代码及例外处理：

```
int main ()
try
```



```

{
    // ... here is our code ...

}
catch(exception& e) {
    // some error reporting
    return 1;
}
catch(...) {
    // some more error reporting
    return 2;
}

```

为了使此 `main()` 编译通过, 我需要定义 `exception`。如果像往常一样包含了头文件 `std lib facilities.h` 或者直接包含标准头文件 `<stdexcept>`, 就会获得 `exception` 的定义。

12.7.2 一个几乎空白的窗口

在这里, 我们不讨论错误处理(参见第 5 章, 特别是 5.6.3 节), 直接进入 `main()` 函数中的图形代码:

```

Point tl(100,100); // top left corner of our window

Simple_window win(tl,600,400,"Canvas");
    // screen coordinate tl for top left corner
    // window size(600*400)
    // title: Canvas
win.wait_for_button(); // display!

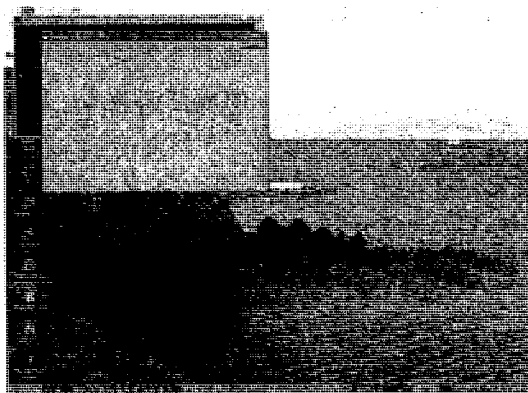
```

这段代码首先创建一个 `Simple_window`, 即一个有“Next”按钮的窗口, 并将它显示在屏幕上。显然, 我们应该包含头文件 `Simple_window.h` 而不是 `Window.h`。在这里, 我们明确给出了窗口在屏幕上的显示位置: 其左上角位于 `Point(100, 100)`。这个位置很接近屏幕的左上角, 但没有过于靠近。很显然, `Point` 是一个类, 其构造函数有两个整型参数, 表示点在屏幕上的 (x, y) 坐标。我们可以将代码写为:

```
Simple_window win(Point(100,100),600,400,"Canvas");
```

然而, 为了便于多次使用点 $(100, 100)$, 我们还是选择给它一个符号名称。600 和 400 分别是窗口的宽度和高度, `Canvas` 是在窗口框上显示的标签。

为了真正将窗口绘制在屏幕上, 我们必须将控制权交给 GUI 系统。我们通过调用 `win.wait_for_button()` 来达到这一目的, 结果如下图所示:



在窗口的背景中, 我们看到了一个笔记本电脑的桌面(已经临时清理过了)。如果你对桌面背景这种不相关的事情感到好奇, 我可以告诉你, 我拍摄照片时正站在安提布的毕加索图书馆附近俯瞰

尼斯湾。隐藏在程序窗口之后的黑色控制台窗口是用来运行我们的程序的。使用控制台窗口不太美观,而且也不是必须的,但当一个尚未调式通过的程序陷入无限循环或无法继续执行时,我们可以通过它来终止程序。如果你仔细观察,会发现我们使用的是微软 C++ 编译器,当然你可以使用其他编译器(如 Borland 或者 GNU)。

在后面的介绍中,我们将去掉程序窗口周围分散注意力的内容,仅仅给出窗口本身如右图所示。窗口的实际尺寸(以像素计算)依赖于屏幕的分辨率。某些屏幕的像素要比其他屏幕更大。



12.7.3 坐标轴

一个几乎空白的窗口没有什么意思,最好给它添加一些信息。希望添加一些什么内容呢?注意,并不是所有的图形都是有趣的或者是关于游戏的,我们将从坐标轴——一种非娱乐性的、有点复杂的图形开始。没有坐标轴的帮助,我们通常难以弄清数据的含义。或许你可以借助伴随的文字,但使用坐标轴要保险得多——人们通常不会阅读文字描述,而且好的图形表示通常会与最初的上下文分离。因此,图形需要坐标轴:

```
Axis xa(Axis::x, Point(20,300), 280, 10, "x axis");    // make an Axis
// an Axis is a kind of Shape
// Axis::x means horizontal
// starting at (20,300)
// 280 pixels long
// 10 "notches"
// label the axis "x axis"

win.attach(xa);    // attach xa to the window, win
win.set_label("Canvas #2");    // relabel the window
win.wait_for_button();    // display!
```

具体操作步骤为:创建坐标轴对象,将其添加到窗口,最后进行显示,如右图所示。可以看到,Axis::x 是一条水平线,其上有指定数量(10 个)的刻度和一个标签“x axis”。通常,标签用于解释坐标轴和刻度的含义。我们通常把 x 轴放在窗口底端附近。在实际应用中,我们更喜欢用符号常量来表示高度和宽度,这样“在底端上方附近”就可以用 y_max-bottom_margin 这样的符号表示,而不是用 300 这样的“魔数”(参见 4.3.1 节和 15.6.2 节)。



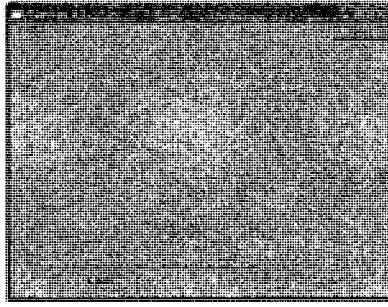
为了帮助识别程序的输出,我们用 Window 的成员函数 set_label() 将该窗口重新标记为“Canvas #2”。

现在,添加一个 y 坐标轴:

```
Axis ya(Axis::y, Point(20,300), 280, 10, "y axis");
ya.set_color(Color::cyan);    // choose a color
ya.label.set_color(Color::dark_red);    // choose a color for the text
win.attach(ya);
win.set_label("Canvas #3");
win.wait_for_button();    // display!
```

我们将 y 轴和标签的颜色分别设置为 cyan 和 dark red。我们认为 x 轴和 y 轴使用不同颜色并不是一个好主意,这里只是为了说明如何设置形状或者其中某个元素的颜色。使用很多种颜色未必是

个好主意，特别是初学者更容易热衷于使用很多颜色。



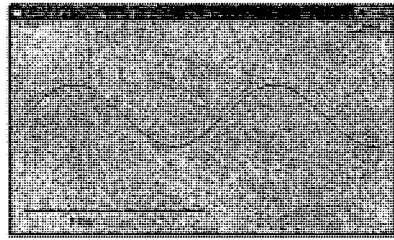
12.7.4 绘制函数图

接下来做什么？现在，我们已经有了一个包含坐标轴的窗口，因此看起来画出一个函数是个好主意。我们创建一个形状来表示正弦函数，并将它添加到窗口：

```
Function sine(sin,0,100,Point(20,150),1000,50,50); // sine curve
// plot sin() in the range [0:100] with (0,0) at (20,150)
// using 1000 points; scale x values *50, scale y values *50

win.attach(sine);
win.set_label("Canvas #4");
win.wait_for_button();
```

在这段代码中，名为 `sine` 的 `Function` 对象使用标准库函数 `sin()` 产生的值绘制一条正弦曲线。我们将在 15.3 节详细讨论如何绘制函数图。现在，你只需要知道绘制函数图时必须给出起始点的位置和输入值集合（值域），并且还需要给出一些信息来说明如何将这内容塞入窗口（缩放），如右图所示。请注意在到达窗口右边界时曲线是如何停止的。



的。当我们绘制的点超出窗口矩形区域时，将被 GUI 系统简单忽略掉，永远不会真正显示出来。

12.7.5 Polygon

函数图是表示数据的一种方法，在第 15 章将会看到更多实例。我们还可以在窗口中绘制不同类型的对象：几何形状。我们使用几何形状来进行图形演示，可以指示用户交互组件（如按钮），通常还能使演示更加生动。`Polygon`（多边形）描述为一个点的序列，这些点通过线连接起来就构成 `Polygon` 类。第一条线连接第一个点到第二个点，第二条线连接第二个点到第三个点……最后一条线连接最后一个点到第一个点：

```
sine.set_color(Color::blue); // we changed our mind about sine's color

Polygon poly; // a polygon; a Polygon is a kind of Shape
poly.add(Point(300,200)); // three points make a triangle
poly.add(Point(350,100));
poly.add(Point(400,200));

poly.set_color(Color::red);
poly.set_style(Line_style::dash);
win.attach(poly);
win.set_label("Canvas #5");
win.wait_for_button();
```

这段代码首先展示了如何改变正弦曲线的颜色。然后，与 12.3 节的例子一样，我们添加了一个三角形，作为一个多边形的例子。然后我们再次设置了颜色，最后设置了线型。`Polygon` 的线都有

“线型”，默认线型为实线，但我们也可根据需要改为虚线、点状线等，如下图所示：

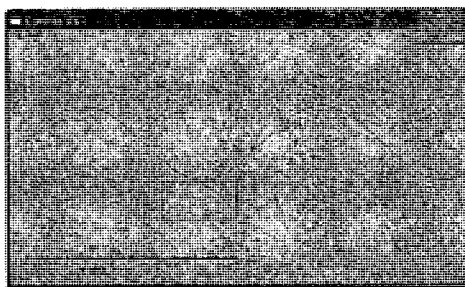
12.7.6 Rectangle

屏幕是矩形，窗口是矩形，一张纸也是一个矩形。实际上，现实世界中有很多形状都是矩形（或者至少是圆角矩形），原因在于矩形是最容易处理的形状。例如，矩形的易于描述（左上角和宽度和高度，或者左上角和右下角，诸如此类）、易于判断一个点在矩形之内还是之外、易于用硬件快速绘制由像素构成的矩形。

与其他封闭的形状相比，大多数高层图形库能够更好地处理矩形。因此，我们将矩形类 `Rectangle` 从多边形类 `Polygon` 中独立出来。一个 `Rectangle` 可以用左上角坐标、宽度和高度来描述：

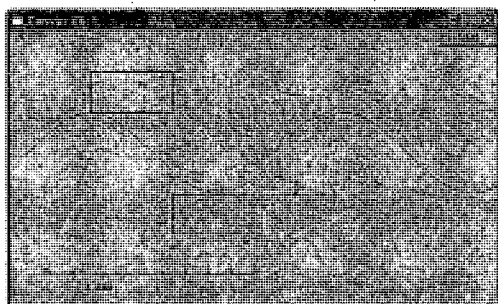
```
Rectangle r(Point(200,200), 100, 50); // top left corner, width, height
win.attach(r);
win.set_label("Canvas #6");
win.wait_for_button();
```

由此可得到下图：



请注意，将位置正确的 4 个点连接起来并不一定得到一个 `Rectangle`。当然，创建一个看起来像 `Rectangle` 的 `Closed_polyline` 是很简单的（你甚至可以创建一个看起来像是 `Rectangle` 的 `Open_polyline`），例如：

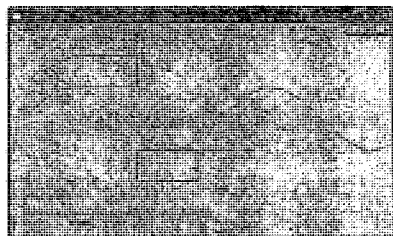
```
Closed_polyline poly_rect;
poly_rect.add(Point(100,50));
poly_rect.add(Point(200,50));
poly_rect.add(Point(200,100));
poly_rect.add(Point(100,100));
```



实际上，`poly_rect` 对应的屏幕图像（`image`）是一个矩形。但内存中的 `poly_rect` 对象并不是一个 `Rectangle` 对象，而且它也不知道有关“矩形”的任何内容。验证这一点的最简单的方法是再添加一个点：

```
poly_rect.add(Point(50,75));
```

矩形是不会有 5 个点的，如右图所示。对于我们分析程序非常重要的一点是：`Rectangle` 不仅仅是在屏幕上看起来像是一个矩形而已，它还应该从根本上保证此形状（几何意义



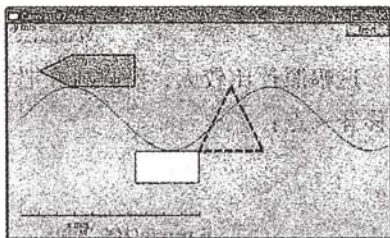
上)始终是一个矩形。这样,我们编写代码时就可以信赖 Rectangle——它确实表示一个矩形,而且保证不会改变为其他形状。

12.7.7 填充

前面绘制形状都是绘制轮廓,我们也可以使用某种颜色“填充”形状:

```
r.set_fill_color(Color::yellow);    // color the inside of the rectangle
poly.set_style(Line_style::dash,4);
poly_rect.set_style(Line_style::dash,2);
poly_rect.set_fill_color (Color::green);
win.set_label("Canvas #7");
win.wait_for_button();
```

我们同时将矩形 (poly) 的线型改为“粗(4 倍粗细)虚线”。类似地,我们也改变了 poly_rect (现在已经不是矩形了)的线型,如右图所示。如果你仔细观察 poly_rect,你会发现轮廓是在填充色上层显示的。

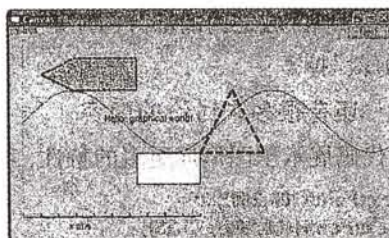


任何封闭的形状都可以填充(参见 13.9 节)。矩形很特殊,它非常容易填充(填充速度也非常快)。

12.7.8 文本

最后,任何一个绘图系统都不可能完全没有简单的文本输出方式——将每个字符看做线的集合来绘制,并保证不会剪切掉字符。我们已经展示了如何为窗口和坐标轴设置标签,我们也使用 Text 对象将文本放置在任意位置:

```
Text t(Point(150,150), "Hello, graphical world! ");
win.attach(t);
win.set_label("Canvas #8");
win.wait_for_button();
```

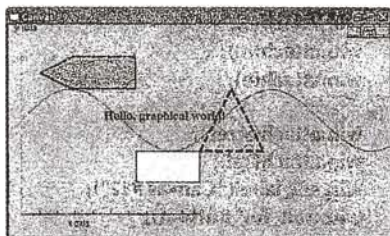


利用此例中的基本图形元素,你可以生成任何复杂、微妙的显示效果。请注意本章所有代码的一个共同特点:没有循环和选择语句,而且所有数据都是“硬编码的”。输出内容只是基本图形元素的简单组合。一旦我们开始使用数据和算法来组合这些基本图形,就可以得到更复杂、更有趣的输出效果了。

我们已经看到如何改变文本的颜色了:坐标轴的标签(参见 12.7.3 节)本身就是一个 Text 对象。此外,我们还可以为文本选择字体和字号:

```
t.set_font(Font::times_bold);
t.set_font_size(20);
win.set_label("Canvas #9");
win.wait_for_button();
```

这段代码将 Text 的字符串“Hello, graphical world!”中的字符放大到 20 号字,字体设置为粗体 Times,如右图所示。



12.7.9 图片

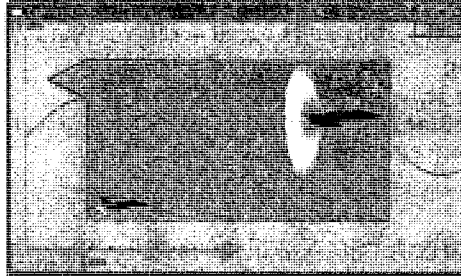
我们还可以从文件中加载图片:

```
Image ii(Point(100,50),"image.jpg");    // 400*212-pixel jpg
win.attach(ii);
```



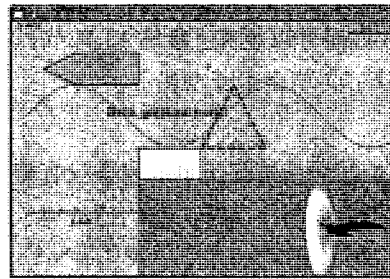
```
win.set_label("Canvas #10");
win.wait_for_button();
```

执行上述代码将在窗口中显示文件名为 image.jpg 的照片(两架飞机正在突破音障):



这幅照片比较大,我们刚好把它放在了文本和图形上层。因此,为了清理窗口,我们将它稍微移开一点:

```
ii.move(100,200);
win.set_label("Canvas #11");
win.wait_for_button();
```



注意,不在窗口区域之内的部分图片是如何被简单忽略掉的。超出窗口区域的内容都会这样被图形系统“剪裁”掉。

12.7.10 还有很多未讨论的内容

下面代码展示了图形库更多的特性,我们在这里不再进行详细解释:

```
Circle c(Point(100,200),50);
Ellipse e(Point(100,200), 75,25);
e.set_color(Color::dark_red);
Mark m(Point(100,200),'x');

ostringstream oss;
oss << "screen size: " << x_max() << "*" << y_max()
    << "; window size: " << win.x_max() << "*" << win.y_max();
Text sizes(Point(100,20),oss.str());

Image cal(Point(225,225),"snow_cpp.gif"); // 320*240-pixel gif
cal.set_mask(Point(40,40),200,150);      // display center part of image

win.attach(c);
win.attach(m);
win.attach(e);

win.attach(sizes);
win.attach(cal);
win.set_label("Canvas #12");
win.wait_for_button();
```

你能猜出这段代码会显示什么内容吗?是不是很容易猜?见右图。代码与屏幕显示内容的关联是很直接的。如果你还未看出这段代码是如何产生这样的输出的,请继续学习后序



章节, 很快就会搞清楚的。请注意我们是如何使用 `stringstream` (参见 11.4 节) 来格式化显示尺寸的文本对象的。

12.8 让图形程序运行起来

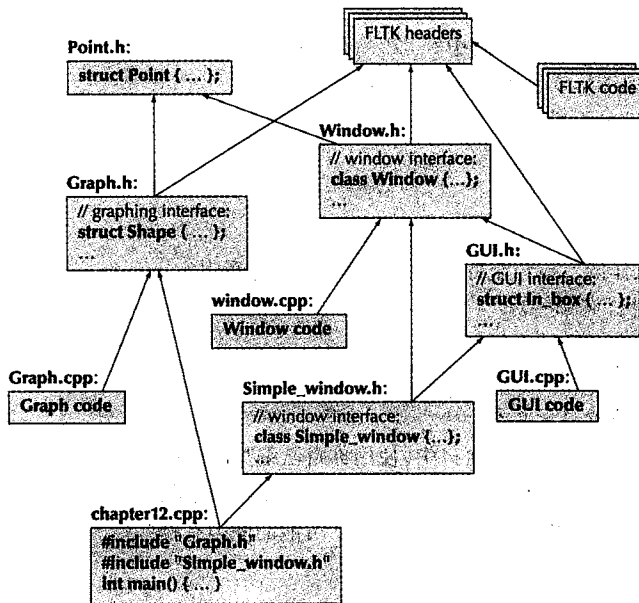
我们已经看到了如何创建窗口以及如何窗口中绘制各种各样的形状。在后续章节中, 我们将会看到这些 `Shape` 类是如何定义的以及它们更多的使用方法。

为了使这个图形程序运行起来, 还需要其他程序的帮助。除了主函数中已有的代码外, 我们还需要编译接口库代码、安装 FLTK 库 (或者所使用的任何 GUI 系统), 并将它与我们的代码正确地链接在一起, 这样才能让这个图形程序运行起来。

我们可以将这个程序看做由 4 个不同的部分构成:

- 我们编写的代码 (`main()` 函数等)
- 接口库 (`Window`、`Shape`、`Polygon` 等)
- FLTK 库
- C++ 标准库

另外, 程序还间接用到了操作系统。忽略了操作系统和标准库之后, 我们的图形代码的组织结构可描述如下:



附录 D 说明了如何让所有这些组成部分一起运转起来。

12.8.1 源文件

我们的图形和 GUI 接口库由 5 个头文件和 3 个代码文件组成:

- 头文件
 - `Point.h`
 - `Window.h`
 - `Simple_window.h`
 - `Graph.h`
 - `GUI.h`

- 代码文件

- Window. cpp
- Graph. cpp
- GUI. cpp

在学习第 16 章之前, 你可以忽略 GUI 源文件。

简单练习

本练习是一个与“Hello, World”程序相同功能的图形程序, 目的是让你熟悉最简单的图形工具。

1. 创建一个空窗口 Simple_window, 尺寸为 600×400 , 标签为 My window, 编译这个程序, 然后编译、链接并运行。注意, 必须以附录 D 描述的方法链接 FLTK 库; 在代码中包含头文件 `#include Graph. h`、`Window. h` 和 `GUI. h`; 并将 `Graph. cpp` 和 `Window. cpp` 加入你的项目中。
2. 逐个添加 12.7 节中的例子, 每添加一个就测试一下。
3. 检查并简单修改每个例子(例如颜色、位置、点的数量等)。

思考题

1. 为什么要使用图形?
2. 什么时候尽量不使用图形?
3. 为什么图形对程序员来说是有趣的?
4. 什么是窗口?
5. 图形接口类(图形库)在哪个名字空间中?
6. 使用图形库实现基本的图形功能需要哪些头文件?
7. 最简单的窗口由哪几部分组成?
8. 什么是最小化窗口?
9. 什么是窗口标签?
10. 如何设置窗口标签?
11. 屏幕坐标系是如何工作的? 窗口坐标系呢? 数学中的坐标系呢?
12. 你能显示的简单“形状”有哪些?
13. 将形状添加到窗口的命令是什么?
14. 你使用哪种基本形状来绘制六边形?
15. 如何在窗口中的某个位置显示文本?
16. 如何将你最好的朋友的照片显示在窗口中(使用你自己编写的程序)?
17. 你创建了一个 Window 对象, 但屏幕上没有显示任何内容, 可能的原因有哪些?
18. 你创建了一个形状, 但窗口中没有显示任何内容, 可能的原因有哪些?

术语

颜色	图形	JPEG	坐标	GUI	线型
显示	GUI 库	软件层	填充颜色	HTML	窗口
FLTK	图像	XML			

习题

我们建议使用 Simple_window 完成下面的练习。

1. 分别用 Rectangle 和 Polygon 绘制矩形, 并将 Polygon 的边设置为红色, Rectangle 的边设置为蓝色。
2. 绘制一个 100×30 的 Rectangle, 并在其内显示文本“Howdy!”。
3. 绘制你名字的每个词的首字母, 高度为 150 个像素, 使用粗线, 每个字母使用不同的颜色。
4. 绘制一个 8×8 的红白交替的国际象棋棋盘。
5. 在矩形周围绘制一个 1/4 英寸宽的红色框, 矩形的高度为屏幕高度的 3/4, 宽度为屏幕宽度的 2/3。

6. 当绘制的 Shape 不能完全放在窗口内时会发生什么现象？当绘制的窗口不能完全置于屏幕内时又会发生什么现象？编写两个程序说明这两种现象。
7. 绘制一个二维的房屋正视图，包括一扇门、两扇窗户、带烟囱的屋顶，可以随意添加其他的细节，比如从烟囱“冒烟”等。
8. 绘制奥林匹克五环旗。如果你记不得颜色了，请查找相关资料。
9. 在屏幕上显示一个图像，例如一个朋友的照片，并使用窗口标题和窗口内的描述文字进行说明。
10. 绘制 12.8 节中的文件结构图。
11. 由内向外绘制一系列正多边形，最里面的是一个等边三角形，外边是一个正方形，再外边是一个正五边形，依此类推。数学专业的人士请注意：让 N 边形的所有顶点都落在 $(N+1)$ 边形的边上。
12. 超椭圆 (superellipse) 是一个由下面的方程定义的二维形状

$$\left| \frac{x}{a} \right|^m + \left| \frac{y}{b} \right|^n = 1; \quad m, n > 0$$

请在互联网上查找超椭圆，以便对这种形状有更感性的认识。编写程序，通过连接超椭圆上的点构成“星状”模式。程序接受参数 a 、 b 、 m 、 n 和 N ，在超椭圆上（由 a 、 b 、 m 和 n 定义）选择 N 个等间隔（按某种“等间隔”的定义）的点，然后将每个点连接到其他一个或多个点（可以用一个参数 k 指出每个点连接的点数，或者直接使用 $N-1$ ，即连接其他所有点）。

13. 设计一种为习题 12 中的超椭圆形状添加颜色的方法。可以为某些线指定一种颜色，其他线使用另一种或另几种颜色。



附言

理想的程序设计是将概念直接表示为程序中的实体。因此，我们通常使用类表示思想，使用类对象表示现实世界的实体，使用函数表示行为和计算。这种思路显然可以用于图形领域。当我们有了概念，例如圆和多边形，就可以在程序中用 Circle 类和 Polygon 类来表示。图形应用的不同寻常之处在于，当我们编写图形程序时，还有机会在屏幕上看到那些类对象。也就是说，程序状态可以直接呈现出来，供我们观察，而在大多数应用中是没这么幸运的。思想、代码和输出的直接对应是图形程序设计如此有吸引力的重要原因。不过，请记住，图形只是体现了在代码中使用类直接表达概念的思想而已。这种思想才是最重要的，它非常有效、通用：我们想到的任何东西都可以在代码中用类、类对象或者一组类来描述。

第13章 图 形 类

“不能改变你的思考方式的语言是不值得学习的。”

——习语

第12章介绍了使用一组简单的接口类可以做哪些图形应用以及如何做。本章将介绍单个接口类，如 Point、Color、Polygon、Open_polyline 等的设计、使用和实现。后续章节将介绍设计一组相关的类的方法及其更多的实现技术。

13.1 图形类概览

图形和 GUI 库提供了大量的特性。“大量”的意思是数百个类，每个类一般都有数十个处理函数。阅读它的说明书、手册或文档有点像阅读一本老式的植物学教材，其中列出了数千种植物的详细信息，而这些植物的分类模糊不清。真是令人沮丧！但也有令人兴奋的一面，浏览最新的图形/GUI 库特性会使你感觉像是一个孩子进入了糖果店，但搞清楚应该从哪里开始，哪些是真正对你有用的，仍旧十分困难。

我们设计这个接口库的目标之一就是减小成熟的图形/GUI 库的复杂性带来的学习难度。我们只提出了 24 个几乎没有任何操作的类，但它们仍能够产生有用的图形输出。另一个紧密相关的目标是通过这些类引入重要的图形和 GUI 概念。现在，你已经能编写程序，将结果显示为简单的图形了。经过本章的学习，你的图形程序设计能力将会超出大多数人最初的期待。经过第 14 章的学习，你将会理解大多数设计技术及其思想，从而能够加深理解，并能够根据需要扩展图形表达能力。为了实现扩展，你既可以向我们的图形/GUI 库中添加新的特性，也可以采用其他 C++ 图形/GUI 库。

重要的接口类如下表所示：

图形接口类	
Color	用于设置线、文本及形状填充的颜色
Line_style	用于设置线型
Point	表示屏幕上和 Window 内的位置
Line	对应屏幕上的一个线段，用两个 Point 对象(端点)来描述
Open_polyline	相连的线段序列，用一个 Point 对象序列来描述
Closed_polyline	与 Open_polyline 类似，唯一的差别是要有一个线段连接最后一个 Point 和第一个 Point
Polygon	所有线段均不相交的 Closed_polyline
Text	字符串文本
Lines	线段集合，用 Point 对的集合来描述
Rectangle	矩形，经过优化，显示快速、便捷
Circle	圆，用圆心和半径定义
Ellipse	椭圆，用圆心和两个轴来描述
Function	函数，画出其一段值域内的图形
Aixs	带标签的坐标轴
Mark	用一个字符(如 x 或 o)标记的点

图形接口类

Marks	带标记(如 x 和 o)的点的序列
Marked_polyline	点被标记的 Open_polyline
Image	图像文件的内容

第 15 章将介绍 Function 和 Axis, 第 16 章介绍主要的 GUI 接口类:

GUI 接口类

Window	屏幕的一个区域, 用来显示图形对象
Simple_window	带有“Next”按钮的窗口
Button	窗口中的一个矩形, 通常带有标签, 可以通过点击它来执行对应函数
In_box	窗口中的一个框, 通常带标签, 用户可在其中输入文本字符串
Out_box	窗口中的一个框, 通常带标签, 用户程序可在其中输出字符串
Menu	Button 向量

源文件组织如下:

图形接口源文件

Point. h	Point
Graph. h	所有其他接口类
Window. h	Window
Simple_window. h	Simple_window
GUI. h	Button 和其他 GUI 类
Graph. cpp	给出 Graph. h 中函数的定义
Window. cpp	给出 Window. h 中函数的定义
GUI. cpp	给出 GUI. h 中函数的定义

除图形类以外, 我们还设计了一个用于保存 Shape 或者 Widget 的容器类:

Shape 或 Widget 的容器

Vector_ref	向量, 其接口便于保存未命名的元素
------------	-------------------

当你阅读下面几节时, 请不要太快。虽然并没有什么困难的内容, 但本章的目的不是向你展示一些漂亮的图片——你每天都会在自己的计算机屏幕上或电视上看到更漂亮的图片, 本章的重点在于:

- 展示代码和它生成的图片之间的关系。
- 让你习惯阅读代码, 思考代码是如何工作的。
- 让你考虑代码的设计, 特别是如何在代码中用类来表示各种概念。为什么这样设计那些类? 还能怎样设计? 在设计中我们做出了很多很多决定, 其中大多数都可以给出同样合理的但不同的决定, 在某些情况下甚至可以彻底不同。

因此, 请不要着急, 否则你将会漏掉一些重要的内容, 因而可能觉得习题很难。

13.2 Point 和 Line

在任何图形系统中, 点(point)都是最基本的组成部分。定义点就是定义我们如何来组织几何

空间。在这里，使用人们习惯的面向计算机的二维布局的点的定义：整数坐标(x, y)。如 12.5 节的描述， x 坐标从 0(屏幕的左边界)到 $x_max()$ (屏幕的右边界)， y 坐标从 0(屏幕的上边界)到 $y_max()$ (屏幕的下边界)。

如头文件 `Point.h` 中定义，`Point` 用一对整数(坐标值)表示：

```
struct Point {
    int x, y;
    Point(int xx, int yy) : x(xx), y(yy) {}
    Point() : x(0), y(0) {}
};

bool operator==(Point a, Point b) { return a.x==b.x && a.y==b.y; }
bool operator!=(Point a, Point b) { return !(a==b); }
```

`Graph.h` 还定义了 `Shape`(将在第 14 章详细介绍)和 `Line`：

```
struct Line : Shape { // a Line is a Shape defined by two Points
    Line(Point p1, Point p2); // construct a Line from two Points
};
```

“`:Shape`”表示 `Line` 是一种 `Shape`，`Shape` 称为 `Line` 的基类(base class)或简称为基(base)。基本上，`Shape` 提供了一些能使 `Line` 的定义更为简单的特性。当我们觉得需要特殊形状，如 `Line` 和 `Open_polyline` 时，我们将会对此进行解释(参见第 14 章)。

`Line` 由两个 `Point` 定义。下面代码创建了 `Line` 对象，并将其绘制出来，我们省略了“基本框架”(`#include` 等语句，参见 12.3 节)：

```
// draw two lines

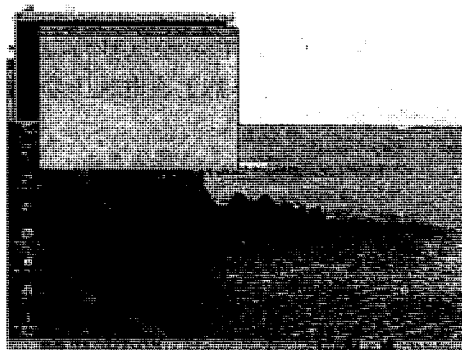
Simple_window win1(Point(100,100),600,400,"two lines");

Line horizontal(Point(100,100),Point(200,100)); // make a horizontal line
Line vertical(Point(150,50),Point(150,150)); // make a vertical line

win1.attach(horizontal); // attach the lines to the window
win1.attach(vertical);

win1.wait_for_button(); // display!
```

执行结果如下：



`Line` 的设计目标就是尽量简单，因此可以工作得很好，对此你不必怀疑：

```
Line vertical(Point(150,50),Point(150,150));
```

这条语句创建一条从点(150, 50)到点(150, 150)的垂直线。当然，我们还不知道 `Line` 的实现细节，但创建 `Line` 对象时不必了解这些信息。实际上，`Line` 的构造函数的实现非常简单：

```

Line::Line(Point p1, Point p2) // construct a line from two points
{
    add(p1);    // add p1 to this shape
    add(p2);    // add p2 to this shape
}

```

也就是说,构造函数只是简单地“添加”了两个点。但是,添加到哪里? Line 又是如何在窗口中绘制出来的? 答案就在 Shape 类中。我们将在第 14 章对此进行介绍,现在你只需要了解, Shape 能够保存定义线的点,知道如何绘制点对构成的线,而且提供了函数 add() 实现将对象添加到 Shape 中。此处的关键点是, Line 的定义非常简单。大多数实现细节都已由“系统”完成了,因此我们可以将精力集中于编写易于使用的类。

从现在开始我们将忽略 Simple_window 的定义和 attach() 的调用。虽然它们对于完整的程序来说是必不可少的框架,但对具体 Shape 的讨论却没有意义。

13.3 Lines

事实上,我们很少仅仅绘制一条线。通常我们思考的问题都是针对包含很多线的对象,如三角形、多边形、路径、迷宫、网格、柱状图、数学函数、数据图等。最简单的“复合图形对象类”是 Lines:

```

struct Lines : Shape { // related lines
    void draw_lines() const;
    void add(Point p1, Point p2); // add a line defined by two points
};

```

Lines 对象是简单的线的集合,每条线由一对 Point 定义。例如,将 13.2 节的 Line 的例子中的两条线作为单个图形对象的组成部分,我们可以这样定义它们:

```

Lines x;
x.add(Point(100,100), Point(200,100)); // first line: horizontal
x.add(Point(150,50), Point(150,150)); // second line: vertical

```

其输出结果很难与 Line 的例子区分开来,如右图所示。能辨别出来这个窗口与 13.2 节中的窗口不同的唯一途径是两者的标签不同。



一组 Line 对象和 Lines 对象中的一组线的区别完全是我们看问题视角的不同。使用 Lines,我们是想表达两条线是联系在一起的,必须一起处理。例如,我们使用单个命令就可以改变 Lines 对象中所有线的颜色。而另一方面,我们却可以为不同的 Line 对象设置不同的颜色。一个更实际的例子的是如何定义网格。网格是由许多等间隔的水平线和垂直线构成的。但是,我们将网格视为一个整体,于是我们将这些水平线和垂直线定义为一个名为 grid 的 Lines 对象的组成部分:

```

int x_size = win3.x_max(); // get the size of our window
int y_size = win3.y_max();
int x_grid = 80;
int y_grid = 40;

Lines grid;
for (int x=x_grid; x<x_size; x+=x_grid)
    grid.add(Point(x,0),Point(x,y_size)); // vertical line
for (int y = y_grid; y<y_size; y+=y_grid)
    grid.add(Point(0,y),Point(x_size,y)); // horizontal line

```

注意, 这里使用 `x_max()` 和 `y_max()` 获得窗口的尺寸, 这也是我们第一个计算显示对象的代码。对于本例, 使用一组 `Line` 对象的方式, 为每条网格线定义一个命名变量, 显然是无法忍受的, 使用一个 `Lines` 对象是更适合的方式。这段代码执行结果如右图所示。

让我们重新回到 `Lines` 的设计。`Lines` 类的成员函数是如何实现的呢? `Lines` 只提供了两种操作, `add()` 函数将一条线(用一个点对定义)添加到要显示的线集合中:

```
void Lines::add(Point p1, Point p2)
{
    Shape::add(p1);
    Shape::add(p2);
}
```

`add(p1)` 前需要加 `Shape::` 限定符, 否则编译器将会调用 `Lines` 类的 `add()` 函数(非法的)而不是 `Shape` 类的 `add()` 函数。

`draw_lines()` 函数绘制 `add()` 定义的线:

```
void Lines::draw_lines() const
{
    if (color().visibility())
        for (int i=1; i<number_of_points(); i+=2)
            fl_line(point(i-1).x, point(i-1).y, point(i).x, point(i).y);
}
```

`Lines::draw_lines()` 一次获取两个点(从点 0 和 1 开始), 并使用底层的画线库函数(`fl_Line()`)绘制两点之间的线。可见性是 `Lines` 类的 `Color` 对象的一个属性(参见 13.4 节), 所以我们必须在画线之前判断其可见性。

如第 14 章所述, `draw_lines()` 是被“系统”调用的。我们不需要检查点的数目是否为偶数, 因为 `Lines` 类的 `add()` 函数每次严格添加两个点。`Shape` 类定义了 `number_of_points()` 函数和 `point()` 函数(参见 14.2 节), 它们以只读方式访问 `Shape` 的点。因为成员函数 `draw_lines()` 不修改形状, 因此将其定义为 `const` 类型(参见 9.7.4 节)。

我们没有为 `Lines` 提供构造函数, 因为以空形状开始, 根据需要逐步添加点的方式更加灵活。我们当然可以为一些简单情况提供构造函数(如包含 1 条线、2 条线和 3 条线)或者任意指定数量的线, 但实际意义并不大。这也是一个基本设计原则, 如果有疑问, 就不要添加功能。因为如果确定了需要这种功能, 可以随时添加, 但是移除一个已经使用的功能是很困难的。

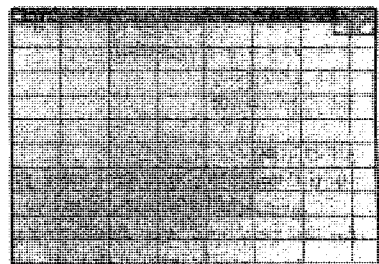
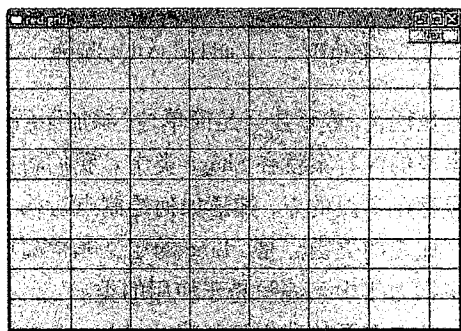
13.4 Color

`Color` 是用于表示颜色的类型, 其使用方法为:

```
grid.set_color(Color::red);
```

该语句将 `grid` 中的线设置为红色, 于是我们得到右面的图。`Color` 定义了颜色的概念, 并给出了一些常用颜色的符号名称:

```
struct Color {
    enum Color_type {
        red=FL_RED,
```



```

    blue=FL_BLUE,
    green=FL_GREEN,
    yellow=FL_YELLOW,
    white=FL_WHITE,
    black=FL_BLACK,
    magenta=FL_MAGENTA,
    cyan=FL_CYAN,
    dark_red=FL_DARK_RED,
    dark_green=FL_DARK_GREEN,
    dark_yellow=FL_DARK_YELLOW,
    dark_blue=FL_DARK_BLUE,
    dark_magenta=FL_DARK_MAGENTA,
    dark_cyan=FL_DARK_CYAN
};

enum Transparency { invisible = 0, visible=255 };

Color(Color_type cc) :c(FL_Color(cc)), v(visible) {}
Color(Color_type cc, Transparency vv) :c(FL_Color(cc)), v(vv) {}
Color(int cc) :c(FL_Color(cc)), v(visible) {}
Color(Transparency vv) :c(FL_Color()), v(vv) {} // default color

int as_int() const { return c; }

char visibility() const { return v; }
void set_visibility(Transparency vv) { v=vv; }

private:
    char v; // invisible and visible for now
    FL_Color c;
};

```

Color 的目标是：

- 隐藏颜色的实现方式：FLTK 的 FL_Color 类型。
- 给定颜色常量的范围。
- 提供一个简单的透明性机制(可见的和不可见的)。

你可以按照以下方式选择颜色：

- 从命名颜色列表中选择，例如 `Color::dark_blue`。
- 从一个小的“调色板”中选择，通过指定 0 ~ 255 之间的一个值，大多数颜色都能很好地在屏幕上显示。例如，`Color(99)` 为暗绿色，代码实例参见 13.9 节。
- 从 RGB(红、绿、蓝)系统中选择一个值，我们不对这种方法进行详细介绍，若需要可以查阅相关资料。特别是，在互联网中搜索“RGB color”，会找到很多资源，比如 www.hypersolutions.org/rgb.html 和 www.pitt.edu/~nisg/cis/web/cgi/rgb.html。(参见习题 13 和 14。)

注意，构造函数可以利用 `Color_type` 或者普通的 `int` 来创建 Color 所有版本的构造函数都将初始化成员变量 `c`。你可能认为 `c` 这个名字太短、太模糊，但由于它只在 Color 内部很小的作用域内使用，不会用于更一般的用途，所以应该没有问题。在我们的设计中，数据成员 `c` 被声明为私有的，以避免用户直接使用它；它的类型被声明为 FLTK 中的 FL_Color 类型——我们不希望将 FL_Color 呈现给用户。但是，将颜色看做 `int` 值(表示其 RGB(或其他颜色系统)值)是很常见的，所以我们提供了 `as_int()` 函数。因为 `as_int()` 函数不会真正改变 Color 对象，故将其定义为常量成员函数。

成员变量 `v` 的取值为 `Transparency::visible` 和 `Transparency::invisible`，表示颜色的透明性(可

见的或者不可见的)。你可能觉得“不可见的颜色”很奇怪,但需要将复合形状的某一部分置为不可见时,这种方式很有效。

13.5 Line_style

在一个窗口中绘制多条线时,可以通过颜色、线型或两者结合将它们区分开来。线型是用来描述线的外形的一种模式,使用方法如下:

```
grid.set_style(Line_style::dot);
```

这条语句将 grid 中的线显示为点状线而非实线,如右图所示。这使得网格看起来变“稀疏”了,更离散了。我们还可以调整线宽(粗细),以使网格线达到我们的喜好和要求。

Line_style 类型的定义如下:

```
struct Line_style {
    enum Line_style_type {
        solid=FL_SOLID,           // -----
        dash=FL_DASH,            // - - - -
        dot=FL_DOT,              // .....
        dashdot=FL_DASHDOT,      // - . - .
        dashdotdot=FL_DASHDOTDOT, // -. -.
    };

    Line_style(Line_style_type ss) : s(ss), w(0) {}
    Line_style(Line_style_type lst, int ww) : s(lst), w(ww) {}
    Line_style(int ss) : s(ss), w(0) {}
    int width() const { return w; }
    int style() const { return s; }

private:
    int s;
    int w;
};
```

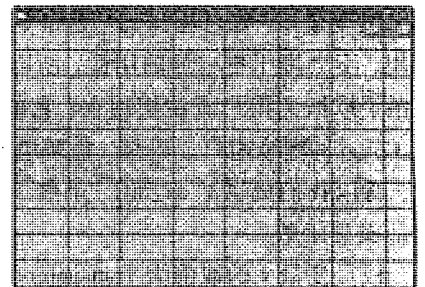
定义 Line_style 所使用的程序设计技术与 Color 的定义完全一样。我们隐藏了 FLTK 使用普通 int 型值表示线型的细节,为什么这样做呢?因为这些实现细节很可能随着库的升级而发生变化。下一个 FLTK 版本完全可能有专门的 Fl_linestyle 类型,我们也完全有可能使用其他 GUI 库而不是 FLTK 来设计接口类。无论哪种情况,我们都不希望我们的代码或者用户的代码充斥着使用普通 int 值表示线型的片段,否则就需要随着库的变化进行大幅度修改。

大多数情况下,我们完全无需担心线型,使用默认线型就可以了(默认宽度和实线)。如果我们没有显式指定线宽,构造函数会设定默认线宽。设定默认值是构造函数所擅长的工作,而恰当的默认值对用户会有很大帮助。

注意,Line_style 有两个分量:模式(如虚线或实线)和宽度(线的粗细)。宽度用整数度量,默认值为 1。我们可以这样来设置粗虚线:

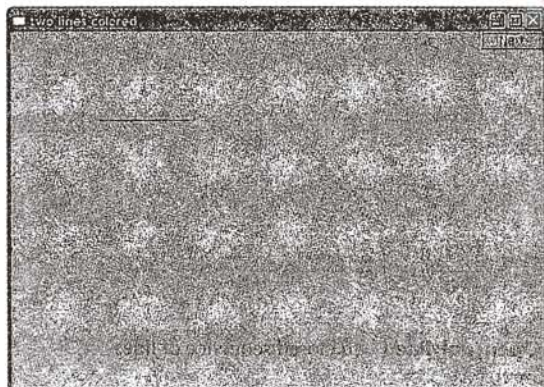
```
grid.set_style(Line_style(Line_style::dash,2));
```

运行结果如右图所示。注意,颜色和线型设定会对形状中所有线起作用,这是将许多线组合为单个图形对象(例如 Lines、Open_polyline 或 Polygon)的好处之一。如果我们想分别控制线的颜色或线型,必须将它们定义为独立的 Line 对象。例如:




```
horizontal.set_color(Color::red);
vertical.set_color(Color::green);
```

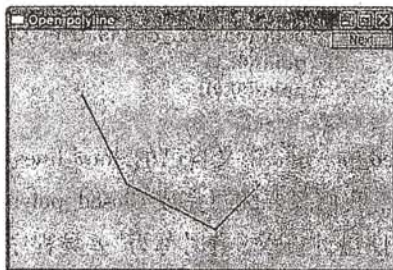
运行结果如下：



13.6 Open_polyline

Open_polyline 是由依次相连的线的序列组成的形状，由一个点的序列定义。poly 一词来源于希腊语，表示“很多”，polyline 是表示由许多线组成的形状的常用术语。例如：

```
Open_polyline opl;
opl.add(Point(100,100));
opl.add(Point(150,200));
opl.add(Point(250,250));
opl.add(Point(300,200));
```



连接所有的点可得到如上图所示的形状。基本上，Open_polyline 不过是我们小时候时的“点连线”游戏的好听一点的说法罢了。

Open_polyline 类的定义如下：

```
struct Open_polyline : Shape { // open sequence of lines
    void add(Point p) { Shape::add(p); }
};
```

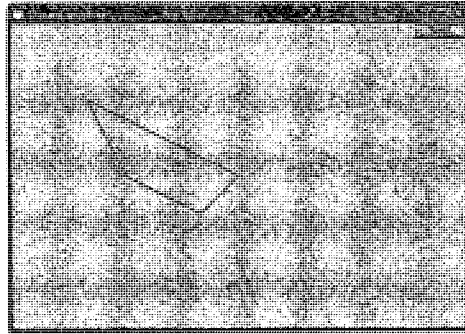
是的，这就是 Open_polyline 的完整定义。从程序文本上看，除了名称和从 Shape 继承来的内容外，Open_polyline 没有定义什么新的东西。唯一的新成员是 add() 函数，它也只是简单地调用 Shape 类的 add() 函数（即 Shape::add()）。我们甚至不必定义 draw_lines()，因为 Shape 类默认情况下会将 add() 函数添加进来的 Point 解释为依次连接的线的序列。

13.7 Closed_polyline

Closed_polyline 与 Open_polyline 很相似，唯一不同之处就是还要画一条从最后一个点到第一个点的线。例如，使用与 13.6 节的 Open_polyline 相同的点构造一个 Closed_polyline：

```
Closed_polyline cpl;
cpl.add(Point(100,100));
cpl.add(Point(150,200));
cpl.add(Point(250,250));
cpl.add(Point(300,200));
```

除了最后的闭合线之外, 结果(当然)与 13.6 节的例子完全相同:



Closed_polyline 的定义为:

```
struct Closed_polyline : Open_polyline { // closed sequence of lines
    void draw_lines() const;
};

void Closed_polyline::draw_lines() const
{
    Open_polyline::draw_lines(); // first draw the "open polyline part"
    // then draw closing line:
    if (color().visibility())
        fl_line(point(number_of_points()-1).x,
                point(number_of_points()-1).y,
                point(0).x,
                point(0).y);
}
```

我们需要为 Closed_polyline 定义自己的 draw_lines() 函数, 来绘制最后一个点到第一个点之间的闭合线。幸运的是, 我们只要编写完成 Closed_polyline 与 Open_polyline 不同的那部分代码即可。这是一种重要的程序设计技术, 有时称为“差异程序设计”——我们只需为派生类(本例中的 Closed_polyline)和基类(本例中的 Open_polyline)的差异编写代码。

那么, 我们如何绘制闭合线呢? 我们使用 FLTK 的画线函数 fl_line() 来完成这一工作。它接受 4 个整型参数, 表示两个点。我们这里再次用到了底层的图形库。但是, 请注意, 与其他例子一样, 我们只是在类的实现中使用了 FLTK, 并没有将它暴露给用户。任何用户代码都无须引用 fl_line() 函数, 或是了解用整数隐式表示点这类细节。这样, 当我们需要时, 就可以用其他 GUI 库替代 FLTK, 而对用户代码几乎不会有任何影响。

13.8 Polygon

Polygon 与 Closed_polyline 非常相似, 唯一的区别是 Polygon 不允许出现交叉的线。例如, 13.7 节中的 Closed_polyline 是一个多边形, 但如果再添加一个点:

```
cpl.add(Point(100,250));
```

运行结果如右图所示。根据经典几何定义, 这个 Closed_polyline 就不是多边形了。那么, 如何定义 Polygon 才能正确利用与 Closed_polyline 之间的关系, 又不违反几何规则? 前文中有一个强烈的暗示: Polygon 是不存在交叉线的 Closed_polygon。换句话说, 我们就可以强调由点建立形状的过程, 如果新添加的 Point 定义的线段不与 Polygon 任何



现有的线相交时, 这样的 `Closed_polyline` 就是 `Polygon`。

由此可定义 `Polygon` 如下:

```
struct Polygon : Closed_polyline { // closed sequence of nonintersecting lines
    void add(Point p);
    void draw_lines() const;
};

void Polygon::add(Point p)
{
    // check that the new line doesn't intersect existing lines
    Closed_polyline::add(p);
}
```

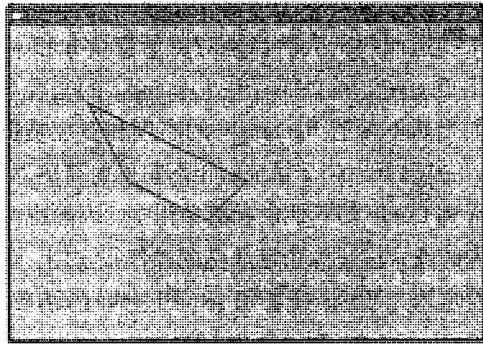
我们继承了 `Closed_polyline` 中 `draw_lines()` 函数的定义, 这不但节省了工作量, 还避免了重复代码。不幸的是, 每次调用 `add()` 时, 我们都需要检查是否有线段交叉。这导致一个低效的(平方阶的)算法——定义一个由 N 个点构成的 `Polygon` 时, 需调用 $N * (N - 1) / 2$ 次 `intersect()` 函数, 因此算法的时间复杂度为 N 的平方阶。在实际应用中, 我们假设 `Polygon` 类只用于顶点数比较少得多边形。例如, 创建一个 24 个 `Point` 的 `Polygon` 需要调用 `intersect()` 函数 $24 * (24 - 1) / 2 == 276$ 次, 这应该是可以接受的, 但如果创建由 2000 个顶点构成的多边形则需要大约 2 000 000 次函数调用, 可能需要寻找更优的算法, 接口可能也需要相应修改。

无论如何, 我们可以这样创建多边形:

```
Polygon poly;
poly.add(Point(100,100));
poly.add(Point(150,200));
poly.add(Point(250,250));
poly.add(Point(300,200));
```

显然, 该代码创建了一个与 13.7 节的 `Closed_polyline` 等价的 `Polygon`, 如右图所示。确保 `Polygon` 真正表示多边形是极其棘手的, `Polygon::add()` 函数中省略的相交检查是整个图形库中最复杂的部分。如果你对高精度几何坐标计算感兴趣, 可以研究一下它的代码。即使解决了相交检查, 多边形判定仍没有彻底解决。考虑创建只有两个点的多边形的请求, 我们显然应该阻止这种情况:

```
void Polygon::draw_lines() const
{
    if (number_of_points() < 3) error("less than 3 points in a Polygon");
    Closed_polyline::draw_lines();
}
```



我们要做的实际上是要保证 `Polygon` 的不变式“这些点表示一个多边形”, 但棘手的是, 只有定义了所有点之后才能验证这个不变式。也就是说, 我们不能在构造函数中建立 `Polygon` 的不变式, 虽然这是最好的方式。将“至少 3 个点”的检测置于 `Polygon::draw_lines()` 函数中也是一种无奈之举(参见习题 18)。

13.9 Rectangle

在屏幕上最常见的形状是矩形, 部分是因为文化(大多数门、窗、照片、墙、书柜、页等都是矩形的), 部分是因为技术(保证坐标位于矩形空间内, 比任何其他形状都简单)。无论如何, 矩

形是如此常见，因而 GUI 系统直接支持矩形，而不是把它们当做 4 个角都是直角的多边形。

```
struct Rectangle : Shape {
    Rectangle(Point xy, int ww, int hh);
    Rectangle(Point x, Point y);
    void draw_lines() const;

    int height() const { return h; }
    int width() const { return w; }
private:
    int h; // height
    int w; // width
};
```

使用两个顶点(左上角和右下角)，或者一个顶点(左上角)和宽度、高度就可以定义矩形。构造函数可以定义如下：

```
Rectangle::Rectangle(Point xy, int ww, int hh)
    : w(ww), h(hh)
{
    if (h<=0 || w<=0)
        error("Bad rectangle: non-positive side");
    add(xy);
}

Rectangle::Rectangle(Point x, Point y)
    : w(y.x-x.x), h(y.y-x.y)
{
    if (h<=0 || w<=0)
        error("Bad rectangle: non-positive width or height");
    add(x);
}
```

每个构造函数都会恰当地对成员变量 h 和 w 进行初始化(使用成员初始化列表，参见 9.4.4 节)，并将矩形左上角点保存在 Rectangle 的基类 Shape 中(使用 add())。此外，还进行了完整性检查：我们当然不希望 Rectangle 的高度和宽度是负数。

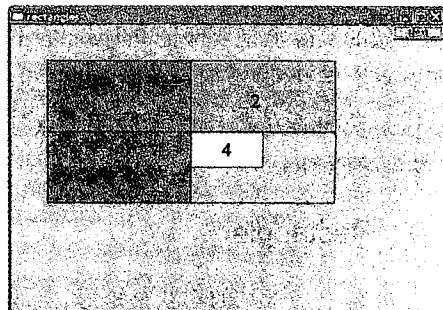
一些图形/GUI 系统对矩形特殊对待的原因之一是，判断哪些像素位于矩形内部的算法要比 Polygon 和 Circle 等其他形状简单得多，因而也快得多。因此，对于矩形，“填充”操作——也就是将区域内的像素设置为指定颜色的操作很常用，而其他形状则较少应用这个操作。我们可以在构造函数中设定填充颜色，或者用 set_fill_color() 函数进行设定(Shape 类提供的颜色相关的操作之一)：

```
Rectangle rect00(Point(150,100),200,100);
Rectangle rect11(Point(50,50),Point(250,150));
Rectangle rect12(Point(50,150),Point(250,250)); // just below rect11
Rectangle rect21(Point(250,50),200,100); // just to the right of rect11
Rectangle rect22(Point(250,150),200,100); // just below rect21

rect00.set_fill_color(Color::yellow);
rect11.set_fill_color(Color::blue);
rect12.set_fill_color(Color::red);
rect21.set_fill_color(Color::green);
```

运行结果如右图所示。当不设定填充颜色时，矩形是透明的，这也是在右图中你可以看到黄色矩形 rect00 的一角的原因。

你可以在窗口内部随意移动形状(参见 14.2.3 节)，例如：



1—蓝色 2—绿色 3—红色 4—黄色

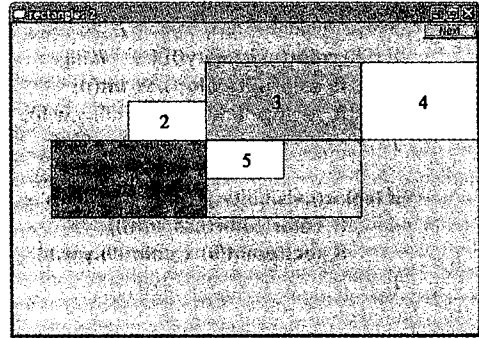
```
rect11.move(400,0);    // to the right of rect21
rect11.set_fill_color(Color::white);
win12.set_label("rectangles 2");
```

运行结果，如右图所示。注意，白色矩形 rect11 只有一部分位于窗口之内，位于窗口之外的部分被“剪裁”掉了，不会在屏幕上显示。

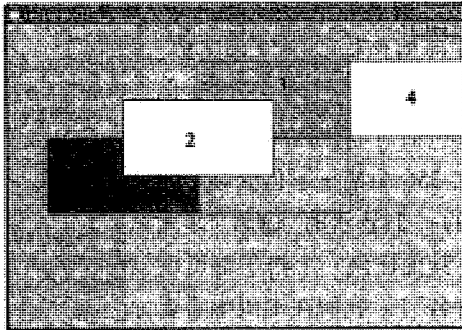
另外请注意形状的层次，一个形状是如何放置在另一个的上层的。这与你将几张纸放在桌子上是一样的，最先放下的那张纸位于最底层。我们的 Window 类(参见附录 E.3)提供了一种重排形状次序的简单方法，可以使用 `Window::put_on_top()` 函数通知窗口将一个形状放在最顶层。例如：

```
win12.put_on_top(rect00);
win12.set_label("rectangles 3");
```

运行结果为：



1—红色 2—黄色 3—绿色 4—白色 5—黄色

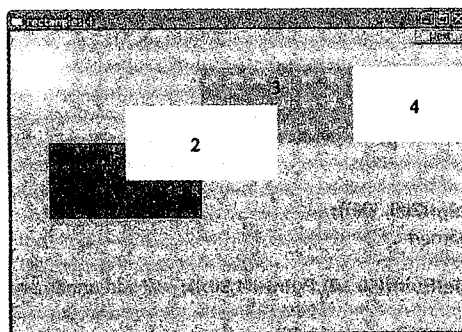


1—红色 2—黄色 3—绿色 4—白色

注意，尽管矩形被填充了某种颜色(除了一个之外)，但仍然可以看到它们的边框。如果不需要，可以将其去掉：

```
rect00.set_color(Color::invisible);
rect11.set_color(Color::invisible);
rect12.set_color(Color::invisible);
rect21.set_color(Color::invisible);
rect22.set_color(Color::invisible);
```

运行结果为：



1—红色 2—黄色 3—绿色 4—白色

注意, 在填充颜色和线的颜色都被设置为 invisible 后, 矩形 rect22 就看不到了。

由于 Rectangle 的 draw_lines() 函数必须处理线的颜色和填充颜色, 因此实现变复杂了:

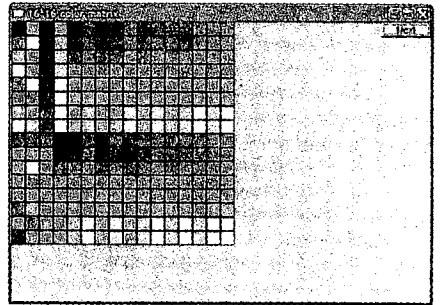
```
void Rectangle::draw_lines() const
{
    if (fill_color().visibility()) { // fill
        fl_color(fill_color().as_int());
        fl_rectf(point(0).x, point(0).y, w, h);
    }

    if (color().visibility()) { // lines on top of fill
        fl_color(color().as_int());
        fl_rect(point(0).x, point(0).y, w, h);
    }
}
```

如你所见, FLTK 提供了绘制填充矩形 (fl_rectf()) 和矩形边框 (fl_rect()) 的功能。在我们的代码中, 默认情况下两者均被绘制。

13.10 管理未命名对象

到目前为止, 我们使用的都是命名图形对象。当处理大量对象时, 这种方法不再可行。例如, 绘制 FLTK 调色板中 256 种颜色构成的比色图表, 即绘制 256 个不同颜色填充的格子, 构成一个 16×16 的颜色矩阵, 来显示相近的颜色值对应什么颜色。首先给出结果, 如右图所示。命名 256 个格子不但繁琐, 而且相当不明智。左上角格子的一个显然的“命名”方式是它在矩阵中的位置 (0, 0), 其他任何一个格子都可以由其坐标 (i, j) 进行标识 (“命名”)。我们在本例中需要做的是找到一种表示对象矩阵的方法。我们考虑过使用 `vector<Rectangle>`, 但实践证明不够灵活。例如, 不同类型未命名对象 (元素) 的集合应该是一种很有用的功能, 但 `vector` 无法实现。我们将在 14.3 节讨论灵活性问题, 这里只给出本例的解决方案: 采用能够保存已经命名和未命名对象的向量类型:



```
template<class T> class Vector_ref {
public:
    // ...
    void push_back(T&); // add a named object
    void push_back(T*); // add an unnamed object

    T& operator[](int i); // subscripting: read and write access
    const T& operator[](int i) const;

    int size() const;
};
```

与标准库 `vector` 的使用方法非常类似:

```
Vector_ref<Rectangle> rect;
Rectangle x(Point(100,200),Point(200,300));
rect.push_back(x); // add named

rect.push_back(new Rectangle(Point(50,60),Point(80,90))); // add unnamed

for (int i=0; i<rect.size(); ++i) rect[i].move(10,10); // use rect
```

我们将在第 17 章解释 `new` 操作符, 在附录 E 中给出 `Vector_ref` 的实现。现在, 知道可以用它保存

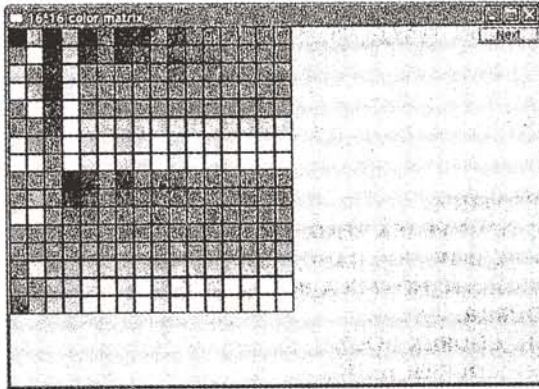
未命名对象就够了。操作符 `new` 后面是类型名称(如 `Rectangle`)，然后是可选初始化列表(如 `(Point(50, 60), Point(80, 90))`)。有经验的程序员可以放心，我们并没有引入内存泄漏问题。

有了 `Rectangle` 和 `Vector_ref` 以后，我们接下来就可以处理颜色了。例如，我们可以这样实现上述具有 256 种颜色的比色图表：

```
Vector_ref<Rectangle> vr;

for (int i = 0; i<16; ++i)
    for (int j = 0; j<16; ++j) {
        vr.push_back(new Rectangle(Point(i*20,j*20),20,20));
        vr[vr.size()-1].set_fill_color(i*16+j);
        win20.attach(vr[vr.size()-1]);
    }
```

这段代码创建了一个保存 256 个 `Rectangles` 的 `Vector_ref`，这些矩形在 `Window` 中排列为 16×16 的矩阵。我们将矩形的颜色设定为 0、1、2、3、4 等。创建一个矩形后，就将其添加到窗口中，显示结果如下：



13.11 Text

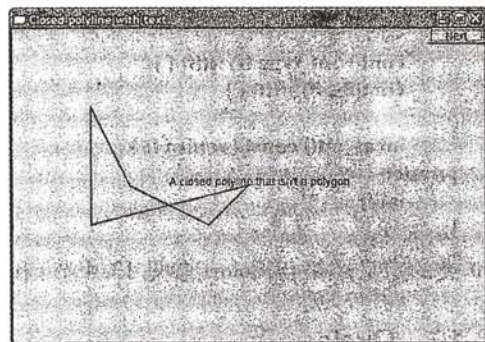
很明显，我们需要在图形显示中添加文本的功能。例如，为 13.8 节中“奇怪”的 `Closed_polyline` 添加标签：

```
Text t(Point(200,200),"A closed polyline that isn't a polygon");
t.set_color(Color::blue);
```

运行结果，如右图所示。一个 `Text` 对象定义了起始位置为 `Point` 的一行文本，其中 `Point` 为文本行的左下角。限制为一行文本的原因是要保证跨系统的可移植性。不要尝试在字符串中放入换行符，在窗口中不一定产生换行效果。字符串流(参见 11.4 节)对于构造 `Text` 对象中的字符串是很有用的(例子参见 12.7.7 和 12.7.8 节)。Text 的定义如下：

```
struct Text : Shape {
    // the point is the bottom left of the first letter
    Text(Point x, const string& s)
        : lab(s), fnt(fl_font()), fnt_sz(fl_size())
    { add(x); }

    void draw_lines() const;
```



```

void set_label(const string& s) { lab = s; }
string label() const { return lab; }

void set_font(Font f) { fnt = f; }
Font font() const { return fnt; }
void set_font_size(int s) { fnt_sz = s; }
int font_size() const { return fnt_sz; }
private:
    string lab;    // label
    Font fnt;
    int fnt_sz;
};

```

因为只有 Text 类知道其字符串是如何存储的，所以 Text 必须有自己的 draw_lines() 函数：

```

void Text::draw_lines() const
{
    fl_draw(lab.c_str(), point(0).x, point(0).y);
}

```

字符颜色的设置类似于形状(如 Open_polyline 和 Circle)中线的颜色的设置方法，你可以用 set_color() 函数选择一种颜色，用 color() 函数获取当前使用的颜色。字号和字体的处理相似，下面列出了一小部分预定义字体：

```

class Font {    // character font
public:
    enum Font_type {
        helvetica=FL_HELVETICA,
        helvetica_bold=FL_HELVETICA_BOLD,
        helvetica_italic=FL_HELVETICA_ITALIC,
        helvetica_bold_italic=FL_HELVETICA_BOLD_ITALIC,
        courier=FL_COURIER,
        courier_bold=FL_COURIER_BOLD,
        courier_italic=FL_COURIER_ITALIC,
        courier_bold_italic=FL_COURIER_BOLD_ITALIC,
        times=FL_TIMES,
        times_bold=FL_TIMES_BOLD,
        times_italic=FL_TIMES_ITALIC,

        times_bold_italic=FL_TIMES_BOLD_ITALIC,
        symbol=FL_SYMBOL,
        screen=FL_SCREEN,
        screen_bold=FL_SCREEN_BOLD,
        zapf_dingbats=FL_ZAPF_DINGBATS
    };

    Font(Font_type ff) :f(ff) {}
    Font(int ff) :f(ff) {}

    int as_int() const { return f; }
private:
    int f;
};

```

Font 类的定义风格与 Color(参见 13.4 节)和 Line_style(参见 13.5 节)的风格是一样的。

13.12 Circle

为了说明世界并不是完全由矩形构成的，我们设计了 Circle 类和 Ellipse 类。Circle 是由圆心和半径定义的：


```

struct Circle : Shape {
    Circle(Point p, int rr);    // center and radius

    void draw_lines() const;

    Point center() const;
    int radius() const { return r; }
    void set_radius(int rr) { r=rr; }
private:
    int r;
};

```

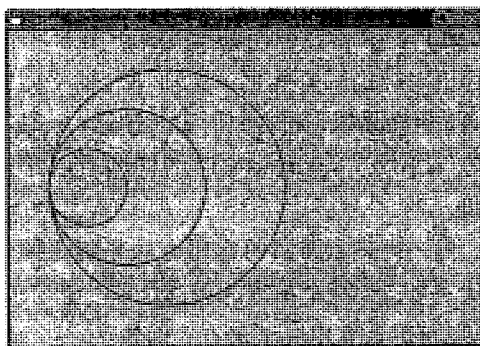
Circle 类的使用方法为：

```

Circle c1(Point(100,200),50);
Circle c2(Point(150,200),100);
Circle c3(Point(200,200),150);

```

上述语句产生圆心在同一水平线上的三个不同半径的圆，运行结果，如右图所示。Circle 类实现的一个特别之处是，它所存储的点不是圆心，而是圆的正方边界的左上角。我们本来可以将两个点都存储起来，但最终选择了存储 FLTK 最优画圆函数所使用的那个。Circle 提供了一个很好的例子，展示了对于同一个概念，一个类如何用来呈现与其实现不同的（可能更好的）视角。



```

Circle::Circle(Point p, int rr)    // center and radius
    :r(rr)
{
    add(Point(p.x-r,p.y-r));    // store top left corner
}

Point Circle::center() const
{
    return Point(point(0).x+r, point(0).y+r);
}

void Circle::draw_lines() const
{
    if (color().visibility())
        fl_arc(point(0).x,point(0).y,r+r,r+r,0,360);
}

```

注意，如何使用 `fl_arc()` 函数绘制圆，其中头两个参数表示左上角，接下来两个参数表示外接矩形的宽度和高度，最后两个参数表示绘制的起止角度。环绕 360 度才绘制出一个圆，但我们也可以用 `fl_arc()` 函数绘制部分圆（椭圆），参见习题 1。

13.13 Ellipse

椭圆与 Circle 类似，但通过长轴和短轴定义，而不是半径。也就是说，定义椭圆需要给出圆心坐标以及从圆心到 x 轴和 y 轴与椭圆交点的距离：

```

struct Ellipse : Shape {
    Ellipse(Point p, int w, int h);    // center, max and min distance from center

    void draw_lines() const;
}

```

```

    Point center() const;
    Point focus1() const;
    Point focus2() const;

    void set_major(int ww) { w=ww; }
    int major() const { return w; }

    void set_minor(int hh) { h=hh; }
    int minor() const { return h; }
private:
    int w;
    int h;
};

```

可以这样使用 Ellipse 类:

```

    Ellipse e1(Point(200,200),50,50);
    Ellipse e2(Point(200,200),100,50);
    Ellipse e3(Point(200,200),100,150);

```

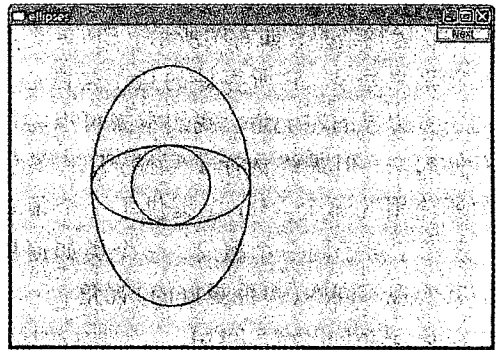
上述语句产生圆心相同、长轴和短轴不同的三个椭圆，运行结果，如右图所示。注意，长轴与短轴相等(`major() == minor()`)的椭圆看起来就是一个圆。

另一种表示椭圆的常用方法是指定两个焦点和从一个点到两个焦点的距离之和。给定一个 Ellipse，可以计算出一个焦点。例如：

```

    Point Ellipse::focus1() const
    {
        return Point(center().x+sqrt(double(w*w-h*h)),center().y);
    }

```



为什么 Circle 不是 Ellipse？从几何角度看，圆都是椭圆，但椭圆不一定是圆。特别地，圆是两个焦点相同的椭圆。我们没有把 Circle 定义为 Ellipse，因为这样会增加一个成员（圆由圆心和半径定义，椭圆由圆心和两个轴定义），带来额外的空间开销。更主要的原因是，如果不禁止 `set_major()` 和 `set_minor()` 函数，就无法将 Circle 定义为 Ellipse。毕竟，如果我们使用 `set_major()` 将长轴设置得与短轴不相等(`major() != minor()`)，就不是一个圆了（从数学家的角度），至少在设置之后就不再是圆了。我们不允许对象的类型发生变化，即一个对象不能在 `major() != minor()` 时是椭圆，而在 `major() == minor()` 时又是圆。但是，能够允许的是一个 Ellipse 对象有时看起来像是圆。另一方面，Circle 永远不会变成两个轴不相等的椭圆。

在设计类时，我们应该小心不要自作聪明，也不要被“直觉”所欺骗，以至于设计出一些毫无意义的“类”来。相反，我们应该注意如何用类表达某些关系密切的概念，不能设计成简单的数据和函数成员的集合。不思考要表达的思想/概念，只是将代码简单地堆积在一起，会制造出我们难以解释和其他程序员难以维护的“黑客代码”。如果你不是利他主义者，请记住“其他程序员”可能就包括几个月之后的你。另外，这种代码也很难调试。

13.14 Marked_polyline

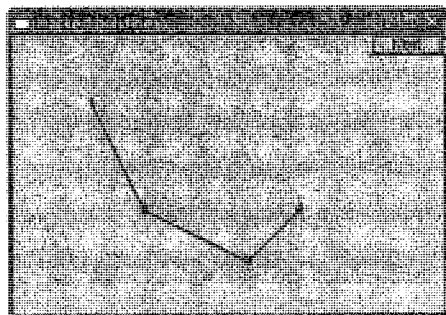
我们常常需要对图中的点做“标记”。画图的一种方式开放的多段线，因此我们只需“标记”开放多段线的每个点即可。Marked_polyline 就能实现这一目的，例如：

```

Marked_polyline mpl("1234");
mpl.add(Point(100,100));
mpl.add(Point(150,200));
mpl.add(Point(250,250));
mpl.add(Point(300,200));

```

运行结果为:



Marked_polyline 的定义为:

```

struct Marked_polyline : Open_polyline {
    Marked_polyline(const string& m) : mark(m) { }
    void draw_lines() const;
private:
    string mark;
};

```

它继承了 **Open_polyline** 类, 因此“免费”实现了对 **Point** 的处理, 我们只需添加处理标记的代码。特别是, **draw_lines()** 函数应修改为:

```

void Marked_polyline::draw_lines() const
{
    Open_polyline::draw_lines();
    for (int i=0; i<number_of_points(); ++i)
        draw_mark(point(i), mark[i%mark.size()]);
}

```

调用 **Open_polyline::draw_lines()** 负责画线操作, 因此我们只需处理标记。我们将标记存储为一个字符串, 按顺序选取其中的字符: 在创建 **Marked_polyline** 时, 使用 **mark[i%mark.size()]** 选择下一个显示的标记字符。其中 % 是模(取余)运算符, 也就是说, 我们循环使用数组 **mark** 来选取标记。这个版本的 **draw_lines()** 函数使用一个小的辅助函数 **draw_mark()**, 完成在给定点实际输出一个字符:

```

void draw_mark(Point xy, char c)
{
    static const int dx = 4;
    static const int dy = 4;

    string m(1,c);
    fl_draw(m.c_str(), xy.x-dx, xy.y+dy);
}

```

其中, 常量 **dx** 和 **dy** 用来使字符居中显示, 字符串 **m** 被初始化为单个字符 **c**。

13.15 Marks

我们有时需要显示不与线关联的标记, 为此我们提供了 **Marks** 类。例如, 我们可以标记上例中用到的 4 个点:

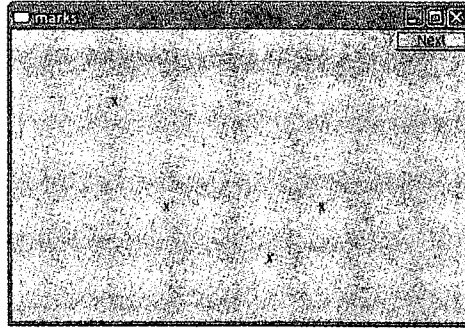
```

Marks pp("x");
pp.add(Point(100,100));

```

```
pp.add(Point(150,200));
pp.add(Point(250,250));
pp.add(Point(300,200));
```

运行结果为：



Marks 的一个明显用途是显示表示离散事件的数据，对这类应用，用线连接各个数据点显然不合理。一个例子是一群人的(身高和体重)数据。

Marks 实际上是简单地将 Marked_polyline 的线设置为不可见(invisible)而实现的：

```
struct Marks : Marked_polyline {
    Marks(const string& m) : Marked_polyline(m)
    {
        set_color(Color(Color::invisible));
    }
};
```

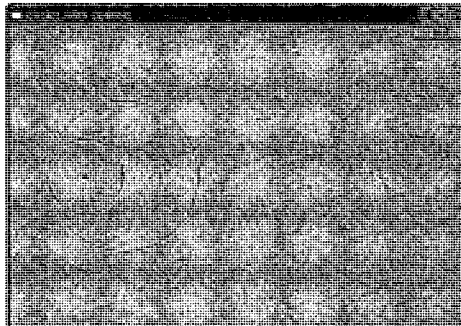
:Marked_polyline(m) 用来 Marked_polyline(作为 Marks 对象的一部分)。这种语法是成员初始化语法的一个变形(参见 9.4.4 节)。

13.16 Mark

Point 只是 Window 中的一个位置而已，不是我们绘制的或是我们能看到的某种东西。若想标记一个孤立的 Point，我们可以像 13.2 节那样使用一对线或者使用 Marks。但这有些繁琐，因此我们实现了 Mark，它由一个点和一个字符构成。例如，可以用它来标记 13.2 节中圆的圆心：

```
Mark m1(Point(100,200),'x');
Mark m2(Point(150,200),'y');
Mark m3(Point(200,200),'z');
c1.set_color(Color::blue);
c2.set_color(Color::red);
c3.set_color(Color::green);
```

运行结果为：



Mark 不过是一个初始化是立刻给定起始点(通常也只有这一个点)的 Marks:

```
struct Mark : Marks {
    Mark(Point xy, char c) : Marks(string(1,c))
    {
        add(xy);
    }
};
```

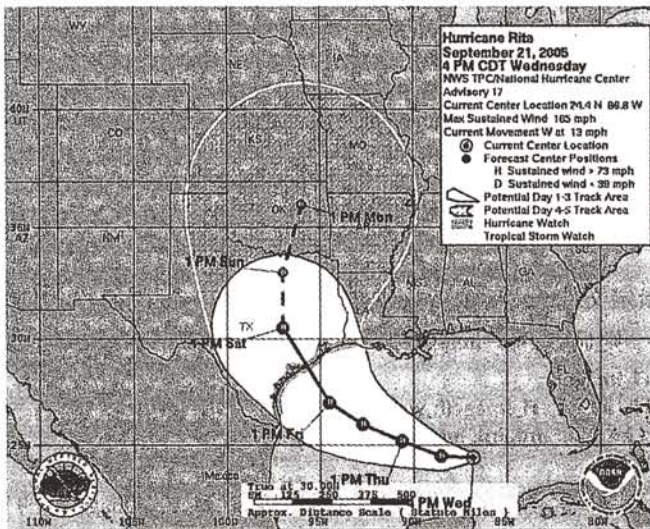
string(1, c) 是字符串 string 类的一个构造函数, 初始化一个仅包含单个字符串 c 的 string 对象。

Mark 的全部作用只是为标记为单个字符的、单个点的 Marks 对象提供了一种简化表示。对于是否值得花力气定义这样一个类? 它是否毫无意义, 只是增加了复杂性和混乱, 还没有一个明确、合理的答案。我们反复推敲过这个问题, 最后还是认为它对用户是有用的, 而实现它的代价很小。

我们为什么使用字符作为“标记”? 我们本可以使用任何小形状, 但字符集合简单丰富, 很适合作为标记。能使用大量不同“标记”来区分不同点通常是很有用的。另外, 像 x、o、+ 和 * 等字符都具有中心对称性。

13.17 Image

平均每台 PC 机保存着数千个图像文件, 而在网络上能找到的图像则数以百万计。很自然地, 我们希望即使是在很简单的程序中也能显示这些图像。例如, 下图(rita_path.gif)是飓风丽塔到达得克萨斯州墨西哥湾的路线图:



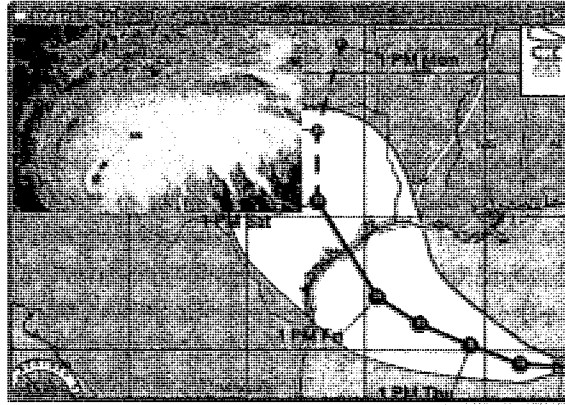
我们可以选择图像的一部分区域, 并加入从太空中拍摄的丽塔的照片(rita.jpg):

```
Image rita(Point(0,0),"rita.jpg");
Image path(Point(0,0),"rita_path.gif");
path.set_mask(Point(50,250),600,400); // select likely landfall

win.attach(path);
win.attach(rita);
```

set_mask() 函数选择要显示图像的某个子图像。在本例中, 它从 rita_path.gif(载入到变量 path) 选择一个 600 × 400 像素大小的图像, 其左上角在 path 中的坐标为 (50, 250)。这种操作十分常见, 所以我们实现了 set_mask 来直接支持它。

形状是按照它们添加的顺序确定层次次序的, 就像将纸放在桌子上一样。由于 path 先于 rita



添加到窗口，所以它在 rita“下层”。

图像编码方式非常多，我们只处理最常用的两种：JPEG 和 GIF：

```
struct Suffix {
    enum Encoding { none, jpg, gif };
};
```

在我们的图形接口库中，图像在内存中用 Image 类对象表示：

```
struct Image : Shape {
    Image(Point xy, string file_name, Suffix::Encoding e = Suffix::none);
    ~Image() { delete p; }
    void draw_lines() const;
    void set_mask(Point xy, int ww, int hh)
        { w=ww; h=hh; cx=xy.x; cy=xy.y; }
private:
    int w,h;    // define "masking box" within image relative to
                // position (cx,cy)
    int cx,cy;
    Fl_Image* p;
    Text fn;
};
```

Image 构造函数打开指定文件，然后按参数或文件后缀名指定的编码格式创建图像。若图像无法显示（如未找到文件），则显示 Bad_image。Bad_image 的定义为：

```
struct Bad_image : Fl_Image {
    Bad_image(int h, int w) : Fl_Image(h,w,0) { }
    void draw(int x,int y, int, int, int, int) { draw_empty(x,y); }
};
```

在图形库中，图像处理是非常复杂的。但我们的图形接口库中的 Image 类的复杂性主要来自构造函数中的文件处理：

```
// somewhat overelaborate constructor
// because errors related to image files can be such a pain to debug
Image::Image(Point xy, string s, Suffix::Encoding e)
    :w(0), h(0), fn(xy,"")
{
    add(xy);

    if (!can_open(s)) { // can we open s?
        fn.set_label("cannot open \""+s+"\"");
        p = new Bad_image(30,20); // the "error image"
        return;
    }
}
```

```

if (e == Suffix::none) e = get_encoding(s);

switch(e) {      // check if it is a known encoding
case Suffix::jpg:
    p = new FI_JPEG_Image(s.c_str());
    break;
case Suffix::gif:
    p = new FI_GIF_Image(s.c_str());
    break;
default:        // unsupported image encoding
    fn.set_label("unsupported file type \""+s+"\"");
    p = new Bad_image(30,20); // the "error image"
}
}

```

我们通过文件名后缀来选择图像对象类型(FI_JPEG_Image 或 FI_GIF_Image)。使用 new 创建对象,并将地址赋予一个指针。这是一个与 FLTK 结构有关的实现细节,这里不详细讨论(参见第 17 章对 new 操作符和指针的讨论)。

现在,我们只需实现 can_open() 函数,来测试是否可以打开指定的文件进行读操作:

```

bool can_open(const string& s)
// check if a file named s exists and can be opened for reading
{
    ifstream ff(s.c_str());
    return ff;
}

```

打开一个文件然后再关闭的方法虽然比较笨拙,但对于区分“不能打开文件”错误和文件中数据格式错误却很有效,具有很好的可移植性。

如果需要,你可以查阅 get_encoding() 函数。其功能抽取给定文件名的后缀,并在已知后缀名表中查找。后缀表是用标准库 map 容器实现的(参见第 18 章)。

简单练习

1. 创建一个 800 × 1000 大小的 Simple_window。
2. 将窗口左侧的 800 × 800 区域绘制为 8 × 8 的网格(因此每个格子的大小为 100 × 100)。
3. 将主对角线上的 8 个格子填充为红色(使用 Rectangle)。
4. 找一个 200 × 200 像素大小的图像(JPEG 或 GIF 格式),在网格中放置它的 3 份拷贝(每个图像占 4 个格子)。如果不能找到恰好为 200 × 200 像素大小的图像,使用 set_mask() 函数从大图像中选择一个 200 × 200 的区域。注意,不要挡住红色的格子。
5. 添加一个 100 × 100 像素大小的图像,当点击“Next”按钮时,将它从一个格子移动到另一个格子。将 wait_for_button() 放在循环中,并编写代码为图像选择下一个格子。

思考题

1. 为什么我们不直接使用商业的或者开源的图形库?
2. 为了实现简单的图形显示,你大约需要使用图形接口库中多少个类?
3. 为了使用图形接口库,需要哪些头文件?
4. 哪些类定义了闭合形状?
5. 为什么不简单地使用 Line 表示所有形状?
6. Point 的参数含义是什么?
7. Line_style 有哪些成员?
8. Color 有哪些成员?
9. 什么是 RGB?

10. 两条 Line 和包含两条线的 Lines 有什么区别?
11. 每种 Shape 都有的属性有哪些?
12. 由 5 个顶点定义的 Close_polyline 有多少条边?
13. 如果你定义了 Shape 但没有添加到 Window 中, 你会看到什么?
14. Rectangle 与包含 4 个 Point(4 个顶点)的 Polygon 有什么区别?
15. Polygon 与 Closed_polygon 有什么区别?
16. 填充和轮廓哪个在更上层?
17. 为什么我们没有定义一个 Triangle 类(毕竟我们定义了 Rectangle)?
18. 在 Window 中怎样移动 Shape?
19. 怎样为 Shape 设置一行文本的标签?
20. 能够为 Text 对象中的文本串设置哪些属性?
21. 什么是字体? 为什么要关心字体?
22. Vector_ref 的作用是什么? 如何使用?
23. Circle 和 Ellipse 的区别是什么?
24. 如果指定的文件不包含图像, 当用该文件显示一个 Image 时会发生什么现象?
25. 如何显示图像的一个子图像?

术语

闭合形状	图像	点	颜色	图像编码	多边形
椭圆	不可见	多段线	填充	JPEG	未命名对象
字体	线	Vector_ref	字号	线型	可见的
GIF	开放形状				

习题

对每个“定义类”的练习, 显示几个对象来验证其正确性。

1. 定义 Arc 类, 绘制部分椭圆。提示: 使用 fl_arc() 函数。
2. 定义由 4 条线和 4 个圆弧组成的 Box 类, 绘制一个圆角矩形。
3. 定义 Arrow 类, 绘制带有箭头的线。
4. 定义函数 n()、s()、e()、w()、center()、ne()、se()、sw() 和 nw()。每个函数接受一个 Rectangle 参数, 返回一个 Point。它们定义了位于矩形的边上和内部的“连接点”。例如, nw(r) 是名为 r 的 Rectangle 的西北(左上)角。
5. 分别为 Circle 和 Ellipse 定义练习 4 给出的函数, 使“连接点”位于图形轮廓上或外部, 但不超出外接矩形。
6. 编写程序绘制一个类似于 12.6 节的类结构图, 如果你第一步定义一个 Box 类表示带有文本标签的矩形, 会简化这个问题。
7. 创建一个 RGB 比色图表(参考 www.lnetcentral.com/rgb-color-chart.html)。
8. 定义 Hexagon 类(hexagon 为正六边形), 构造函数的参数为中心和从中心到每个角的距离。
9. 用 Hexagon 铺贴窗口的一部分区域(至少使用 8 个六边形)。
10. 定义 Regular_polygon 类, 构造函数的参数为中心、边数(>2)和从中心到每个角的距离。
11. 绘制一个 300 × 200 像素大小的椭圆, 然后以圆心为原点, 绘制长度分别为 400 和 300 像素的 x 轴和 y 轴。标记椭圆的两个焦点; 标记椭圆边上不在坐标轴上的一个点, 并绘制连接焦点到该点的两条线。
12. 绘制一个圆, 然后沿圆周移动一个标记(每按一次“Next”按钮, 标记移动一段距离)。
13. 绘制 13.10 节的颜色矩阵, 但不显示每种颜色的边界线。
14. 定义直角三角形类, 并使用 8 个不同颜色的直角三角形绘制一个八边形。
15. 用一些小的直角三角形铺贴窗口。
16. 用六边形重做习题 15。

17. 用一些不同颜色的六边重做习题 16。
18. 定义 Poly 类表示多边形, 并在构造函数中判断给定点是否真的构成一个多边形。提示: 需给定点作为构造函数的参数。
19. 定义 Star 类。其中一个参数为点的数目。使用不同数量的点、不同颜色的边、不同的填充颜色绘制一些星形图。



附言

第 12 章讲解了如何使用图形库的类。本章使我们上升到程序员“食物链”的更上一层: 除了使用工具以外, 还能设计工具。

第14章 设计图形类

“实用的，持久的，优美的。”

——Vitruvius

图形相关的这些章节有两个目的：我们希望为信息显示提供有用的工具，同时我们还希望通过一系列图形接口类来说明一般的设计与实现技术。特别地，本章介绍接口设计的思想和继承的概念。为此，我们不得不先介绍一些和面向对象程序设计直接相关的语言特性：类派生、虚函数和访问控制。我们不认为能孤立于使用 and 实现来讨论设计，所以我们关于图形类设计的讨论是相当具体化的，或许你应该把本章看做“图形类的设计与实现”。

14.1 设计原则

我们的图形接口类的设计原则是什么？首先：这是一个什么类别的问题？什么是“设计原则”？我们为什么要考虑这些设计原则，而不是直接继续考虑如何生成图形这类重要的问题呢？

14.1.1 类型

图形是一个很好的应用领域的例子。因此，我们所关注的是如何为（像我们一样的）程序员提供一组基本的应用程序概念和工具，本章就给出了这样一个例子。如果我们的代码以混乱、不一致、不完整或其他不好的方式呈现这些概念，生成图形输出的难度就会增大。希望我们的图形类能够降低程序员学习和使用的难度。

我们的程序设计理念是用代码直接描述应用领域概念。这样，如果你理解应用领域，你就能理解代码，反之亦然。例如：

- Window——由操作系统负责管理的窗口。
- Line——一条线，就如你在屏幕上所见。
- Point——一个坐标点。
- Color——颜色，就如你在屏幕上所见。
- Shape——所有形状的统一——以我们的图形/GUI 视角来看待世界时。

最后一个例子 Shape 与其他例子不同，它是一个一般性的、纯抽象的概念。我们永远无法在屏幕上看到一个“一般形状”；我们只能看到线、六边形这样的具体形状。这一点已经反映在我们的类型定义中：你可以尝试创建一个 Shape 变量，编译器将会阻止你。

我们的图形接口类构成一个库，这些类经常被组合在一起使用。它们给出了一个示例，当你定义描述其他图形形状的类型时，可以作为参考。这些类也可以作为基本组件，供你来构造描述其他形状的复杂的类。我们并不是仅仅定义了一些无关的类的集合，所以不能孤立地为每个类进行设计。这些类一起提供了一个如何生成图形的视图，我们必须确保这个视图是相当优雅和一致的。考虑到我们的库的规模，以及图形应用领域的庞大，我们显然不能对它的完整性有什么期望。相反，我们的目标是简洁性和可扩展性。

事实上，没有类库能直接对其应用领域的各个方面进行建模。这不仅是不可可能的，而且是毫无意义的。考虑编写一个用于显示地理信息的库，你希望显示植被吗？国家、州或其他的行政边

界呢？道路系统呢？铁路呢？河流呢？突出显示社会和经济数据吗？温度和湿度的季节性变化呢？大气层中风模式呢？航空线路呢？需要标记学校的地点吗？快餐店的地点呢？地方景点呢？对于一个全面的地理应用来说，“这些都要！”可能是一个很好的答案。但对于简单的图形显示程序，显然不是。对于一个支持这类地理应用的库来说，包含所有上述功能可能是一个不错的方案。但是这样的库就“全面”了吗？它不太可能还涵盖其他图形应用，例如徒手绘图、编辑照片图像、科学计算可视化以及航空器控制显示等。

所以，如往常一样，我们必须确定对我们来说什么是最重要的。对于图形库设计，就是要决定我们希望做好哪种图形/GUI。试图做好所有事情通常会走向失败。好的库会从一个特定的角度直接、清晰地建模其应用领域，强调应用的某些方面，对其他方面则不太关注。

我们提供的类都是用于简单的图形和简单的图形用户界面，它们主要针对那些需要图形化输出数值计算/科学计算/工程计算等应用的数据的用户。你可以在这些类的基础之上创建自己的类。如果这还不够，我们已经在实现中提供了足够多的 FLTK 细节，如果你需要，可以从中找到如何更直接地使用它（或者是一个类似的完善的图形/GUI 库）的方法。不过，如果你决定了这样一条路线，请首先掌握第 17 章和第 18 章的内容。这两章包括一些指针和内存管理的相关内容，这都是直接使用大多数图形/GUI 库所必需的。

我们的图形/GUI 库的一个关键设计决策是提供大量的“小”类和较少的操作。例如，我们提供了 `Open_polyline`、`Closed_polyline`、`Polygon`、`Rectangle`、`Marked_polyline`、`Marks` 和 `Mark`，而不是带有很多参数和操作的单一类（可能命名为“`polyline`”），能通过这些参数和操作指定一个对象是哪一种多边形，甚至可能将一种多边形变化为另一种多边形。这种思路的极致就是只提供一个 `Shape` 类，所有形状都归为 `Shape` 的一种情况。我们认为使用很多小类能够更加直接、更加有效地建模我们的图形领域。一个提供“所有形状”的单一类，不具备一个能帮助理解、调试以及提高性能的框架，会使用户对数据和操作感到混乱。

14.1.2 操作

我们为每个类提供了最少的操作。我们的设计理念是用最小的接口来实现我们想做的事情。当我们需要更大的便利性时，可以通过增加非成员函数或新的类来实现。

我们希望所有类的接口有一致的风格。例如，在不同的类中，执行相似操作的所有函数有相同的函数名，接受相同类型的参数，还可能要求这些参数的顺序也相同。考虑设计这样一个构造函数：形状需要一个位置，因此构造函数接受一个 `Point` 作为第一个参数：

```
Line ln(Point(100,200),Point(300,400));
Mark m(Point(100,200),'x');           // display a single point as an "x"
Circle c(Point(200,200),250);
```

所有处理点的函数都使用 `Point` 类来表示点，这看起来是很显然的方式，但是很多类库都采用了多种风格的混合。例如，想象一个简单的画线函数，就可以有两种不同的风格：

```
void draw_line(Point p1, Point p2);      // from p1 to p2 (our style)
void draw_line(int x1, int y1, int x2, int y2); // from (x1,y1) to (x2,y2)
```

我们甚至可以同时允许这两种风格，但是出于一致性以及改进类型检查和可读性的考虑，我们选择第一种方式。一致地使用 `Point` 类还会避免将坐标和其他一般整数对（如宽度和高度）混淆。例如，考虑下面代码：

```
draw_rectangle(Point(100,200), 300, 400); // our style
draw_rectangle(100,200,300,400);           // alternative
```

第一个调用利用 `Point`、宽度和高度绘制了一个矩形，我们可以很容易地推断出这些参数的含

义。但是第二个调用呢？矩形是由(100, 200)和(300, 400)这两个点定义的吗？还是由一个点(100, 200)和宽度 300 以及高度 400 定义的呢？或者是完全不同的其他形式(对某些人可能是合理的形式)？而一致地使用 Point 类可以避免这种混淆。

顺便说一下，如果一个函数需要一个宽度值和一个高度值，那么实参总是按照这样的顺序给出(就像我们总是先给出 x 坐标，再给出 y 坐标一样)。在这种微小细节上保持一致性，会极大地方便使用，减少运行时错误。

逻辑上，等价的操作应该有相同的名字。例如，对任何类型的形状，所有添加点、线等的函数都叫做 add()，所有画线函数叫做 draw_lines()。这种一致性能帮助我们记忆(只需要记住更少的细节)，同时在我们设计新类的时候也能给予我们帮助(“按常规进行即可”)。有时候，这种一致性甚至允许我们编写能用于很多不同类型的代码，因为这些类型上的操作有着同样的模式。这种代码称为泛型程序，参见第 19 ~ 21 章。

14.1.3 命名

逻辑上，不同的操作应该有不同的名字。这看起来似乎是很显然的，但是请考虑：为什么我们将一个 Shape“添加”(attach)到一个 Window 中，但却将一个 Line“加入”(add)一个 Shape 中呢？两个操作都是“将某物放到某物之中”，那么这种相似性是不是应该反映为相同的名字呢？不是！这种相似性背后隐藏了一个根本的不同点。考虑如下代码：

```
Open_polyline opl;
opl.add(Point(100,100));
opl.add(Point(150,200));
opl.add(Point(250,250));
```

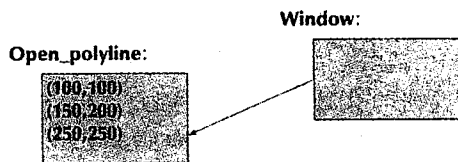
这里，我们将 3 个点加入 opl 中。在 add()调用完成之后，形状 opl 就不再关心“我们的”点了，而是自己为每个点维护一份副本。事实上，我们很少保存点的副本——我们将这个工作交给形状。而另一方面，考虑下面代码：

```
win.attach(opl);
```

这里，我们建立了一个从窗口 win 到我们的形状 opl 的连接；win 不会为 opl 生成一个备份——它仅仅是保存 opl 的一个引用。因此，在 win 使用 opl 期间，保证 opl 可用是我们的责任而不是 win 的责任。也就是说，在 win 使用 opl 的时候，我们不能让 opl 离开其作用域。我们可以更新 opl，win 下一次绘制 opl 时，所做的更改就会显示在屏幕上。attach()和 add()的区别如右图所示。基本上，add()的参数传递采用传值方式(拷贝副本)而 attach()采用传引用方式(共享单一对象)。我们可以选择将图形对象拷贝到 Window 中，但那将是一个完全不同的程序设计模型，在那种模型中确实应该使用 add()而不是 attach()。但在当前的模型中，我们只是将图形对象“添加”到 Window 中。这种模型有一些重要的暗示。例如，我们不能这样做：建立一个对象，将其添加到窗口中，接着将对象销毁，然后还希望程序能继续正常工作：

```
void f(Simple_window& w)
{
    Rectangle r(Point(100,200),50,30);
    w.attach(r);
} // oops, the lifetime of r ends here

int main()
```



```

{
    Simple_window win(Point(100,100),600,400,"My window");
    //...
    f(win);    // asking for trouble
    //...
    win.wait_for_button();
}

```

当我们已经退出 `f()` 函数, 运行到 `wait_for_button()` 的时候, `win` 所引用和显示的对象 `r` 已经不存在了。在第 17 章中, 我们将展示如何使函数内创建的对象在函数返回后还继续存在。但现在, 我们还是要避免将那些生命期在 `wait_for_button()` 之前就结束的对象添加到窗口。`Vector_ref`(参见 13.10 节和附录 E.4) 可以帮助我们解决这个问题。

注意, 如果我们将 `f()` 的 `Window` 参数声明为 `const` 引用类型(如 8.5.6 节推荐的那样), 编译器会阻止我们犯这类错误: 我们不能 `attach(r)` 到一个 `const Window`, 因为 `attach()` 需要修改 `Window` 对象, 以便记录 `r`。

14.1.4 可变性

当设计一个类时, “谁可以修改其数据(描述)?”以及“如何修改?”是我们必须回答的关键问题。我们试图保证只有类自身能够修改其对象的状态。`public/private` 间的区别是实现这一效果的关键, 但我们将给出使用更加灵活/微妙的机制(`protected`)的例子。这意味着我们不仅仅是为类提供一个数据成员, 比如一个名为 `label` 的 `string` 对象; 我们还必须考虑在构造之后是否允许修改它, 以及如果允许的话, 如何修改。我们还必须决定非成员函数是否需要读取 `label` 的值, 以及如果需要的话, 如何读取。例如:

```

struct Circle {
    //...
private:
    int r;    // radius
};

Circle c(Point(100,200),50);
c.r = -9;    // OK? No — compile-time error: Circle::r is private

```

正像你可能在第 13 章已经注意到的那样, 我们决定阻止对大部分数据成员的直接访问。不直接暴露数据成员, 使我们有检查那些“愚蠢”的数据, 比如一个半径为负数的 `Circle` 对象。出于实现简单的考虑, 我们只是进行了有限的检查, 所以要小心处理你的数据。我们决定不进行一致的、全面的检查, 一方面是希望保持代码的简洁, 另一方面是因为用户(你、我)提供的“愚蠢”数据只会在屏幕上绘制出乱七八糟的图像, 而不会破坏珍贵的数据。

我们将屏幕(可看做一组 `Window`)当做一个纯粹的输出设备。我们可以在屏幕上显示新对象以及移除旧的对象, 但不会向“系统”请求他人绘制的图像的信息, 我们能获取的信息只来自自己创建并表示图像的数据结构。

14.2 Shape 类

`Shape` 类是一个一般概念, 表示可显示在屏幕上 `Window` 中的对象:

- `Shape` 是一个概念, 将图形对象与 `Window` 抽象关联起来, 而 `Window` 提供了操作系统和物理屏幕之间的联系。
- `Shape` 是一个类, 可以处理画线所用的颜色和线型。为了实现这一功能, `Shape` 中保存了一个 `Line_style` 和一个 `Color`(用于线型和填充)。

- Shape 可以包含一个 Point 序列，以及绘制这些点的基本方法。

经验丰富的设计者会意识到，一个处理三方面工作的类很可能会出现。但是，我们这里需要比一般解决方案更简单的方式，因此还是选用了这种设计策略。

我们首先给出完整的类，然后再讨论它的实现细节：

```
class Shape {    // deals with color and style and holds sequence of lines
public:
    void draw() const;           // deal with color and draw lines
    virtual void move(int dx, int dy); // move the shape +=dx and +=dy

    void set_color(Color col);
    Color color() const;

    void set_style(Line_style sty);
    Line_style style() const;

    void set_fill_color(Color col);
    Color fill_color() const;

    Point point(int i) const;      // read-only access to points
    int number_of_points() const;

    virtual ~Shape() {}
protected:
    Shape();
    virtual void draw_lines() const; // draw the appropriate lines
    void add(Point p);               // add p to points
    void set_point(int i, Point p);  // points[i]=p;
private:
    vector<Point> points;           // not used by all shapes
    Color lcolor;                   // color for lines and characters
    Line_style ls;
    Color fcolor;                   // fill color

    Shape(const Shape&);            // prevent copying
    Shape& operator=(const Shape&);
};
```

这是一个相对复杂的类，用以支持各种各样的图形类以及表示屏幕上形状的一般概念。然而，它仍然只有 4 个数据成员和 15 个成员函数。而且，这些函数都较为简单，因此我们可以将注意力集中在设计方面。在本节的剩余部分中，我们将逐个研究这些类成员，并解释它们在设计中的作用。

14.2.1 一个抽象类

考虑 Shape 类的第一个构造函数：

```
protected:
    Shape();
```

构造函数是 protected 的，这意味着只有 Shape 类的派生类可以直接使用它（使用：Shape 符号）。换句话说，Shape 只能用做其他类（如 Line 和 Open_polyline）的基类。“protected：”用于构造函数的目的是：保证我们不直接创建 Shape 对象。例如：

```
Shape ss;    // error: cannot construct Shape
```

Shape 被设计为只能当做一个基类。在这种情况下，如果我们允许直接创建 Shape 对象，不会发生什么特别不好的事情；但由于可以限制性使用，我们仍然保留了修改 Shape 对象的权限，这使得

Shape 类不适于直接使用。同样，通过禁止直接创建 Shape 对象，我们直接实现了这样一种思想：不能创建/显示一般性的形状，而只能创建/显示特定的形状，例如 Circle 或者 Closed_polyline。仔细思考一下这一思想！一个形状看起来是什么样子？唯一合理的回答是反问：“是什么形状？”我们通过 Shape 类所描述的形状概念是一个抽象的概念。这是一种重要的、很常用也很有用的设计思想，因此我们不希望在程序中实践这一思想时打折扣。构造函数可以定义如下：

```
Shape::Shape()
: lcolor(fl_color()),           // default color for lines and characters
  ls(0),                       // default style
  fcolor(Color::invisible)     // no fill
{
}
```

这是一个默认构造函数，所以它将成员设置为默认值。再次强调一下，实现中使用的底层库 FLTK 完成了实际工作。然而，这里对 FLTK 的使用并没有直接提及 FLTK 的颜色和风格的概念，它们只是作为 Shape、Color 和 Line_style 类实现的一部分。vector<Points> 的默认值为空向量。

如果一个类只能被用做基类，它就是一个抽象类。另一种更常用的定义抽象类的方法称为纯虚函数(pure virtual function)，参见 14.3.5 节。与抽象类相对的是具体类，即可以创建对象的类。注意，抽象和具体是一对非常简单的技术词汇，我们可能每天都会用到它们来表示区别。我们可能去商店买一台照相机，但是不会只向售货员要一台“照相机”。照相机是什么牌子的？具体是什么型号？单词“照相机”是一个通称；它代表一个抽象的概念。而“Olympus E-3”代表具体的一类照相机，而我们(花费一大笔钱)可以获得它的一个特定实例：一个具有唯一序列号的特定的照相机。所以，“照相机”更像一个抽象类(基类)，“Olympus E-3”更像一个具体类(派生类)，而我手中的真实的照相机(如果我买了它)则更像是一个对象。

声明“virtual ~Shape(){}”定义了一个虚析构函数。我们现在还不会用到它，所以将在 17.5.2 节进行介绍，在那里我们介绍如何使用它。

14.2.2 访问控制

Shape 类将所有数据成员均声明为 private：

```
private:
    vector<Point> points;
    Color lcolor;
    Line_style ls;
    Color fcolor;
```

因为 Shape 类的数据成员被声明为 private，因此我们需要为它们提供访问函数。访问函数的设计有多种风格，我们选择了一种较为简单、方便、易读的方式。如果有一个成员代表一个属性 X，我们可以提供一对函数 X() 和 set_X() 分别用于该成员(属性)的读和写。例如：

```
void Shape::set_color(Color col)
{
    lcolor = col;
}

Color Shape::color() const
{
    return lcolor;
}
```

这种风格最主要的不便之处在于不能将成员变量和读取函数设定为相同的名字。像以往一样，我们将最方便的名字赋予函数，因为它们是公共接口的一部分，而私有变量的命名就不那么重要

了。注意，我们用 `const` 指出读取函数不能修改 `Shape` 对象（参见 9.7.4 节）。

`Shape` 类保存一个名为 `points` 的 `Point` 向量，`Shape` 负责它的维护，用来支持其派生类。我们提供了将 `Point` 对象添加到 `points` 中的函数 `add()`：

```
void Shape::add(Point p)    // protected
{
    points.push_back(p);
}
```

`points` 初始时当然应该是空的。我们决定为 `Shape` 提供一个完整的功能接口，而不是让用户（即使是 `Shape` 的派生类的成员函数）直接访问数据成员。对于某些人来说，提供功能接口是非常正常的，因为他们觉得将类的成员设计为公有（`public`）是不好的设计。而对于另一些人，我们的设计看起来过于严格了，因为我们甚至不允许派生类的成员函数直接对数据成员进行访问。

一个派生自 `Shape` 的形状，比如 `Circle` 和 `Polygon`，是了解点（`points`）的含意的。基类 `Shape` 则并不“理解”这些点，它只是存储它们。因此，派生类需要控制如何添加点。例如：

- `Circle` 和 `Rectangle` 不允许用户添加点，因为添加点没有任何意义。一个矩形加一个额外的点又是什么呢（参见 12.7.6 节）？
- `Lines` 只允许添加成对的点（而不是一个单独的点；参见 13.3 节）。
- `Open_polyline` 和 `Marks` 允许添加任意多个点。
- `Polygon` 只允许通过具有相交性检查功能的 `add()` 函数来添加点（参见 13.8 节）。

我们将 `add()` 设计为 `protected`（即只能从派生类进行访问），保证由派生类来控制如何添加这些点。如果 `add()` 函数为 `public`（任何人都可以添加点）或者 `private`（只有 `Shape` 可以添加点），就会使得实际功能无法符合我们对形状的设想。

同样，我们将 `set_point()` 设计为 `protected`，即只有派生类能知道点的含意是什么以及是否可以在不违反不变式的前提下修改它。例如，如果我们有一个 `Regular_hexagon` 类，定义为 6 个点的集合，即使只改变一个点也有可能使图形不再是一个“正六边形”。而另一方面，如果改变四边形的一个点，其结果仍然会是一个四边形。事实上，在示例类和代码中，我们并没有发现有对 `set_point()` 函数的需求，因此 `set_point()` 在这里只是为了保证我们能够读取和设置 `Shape` 的每个属性的设计原则仍旧成立。例如，如果要实现一个 `Mutable_rectangle` 类，我们可以从 `Rectangle` 类派生，并且提供更改变点的操作。

我们将存放 `Point` 的向量 `points` 设计为 `private`，以保护它不会被意外地修改。为了能使用它，我们还需要提供成员函数实现对它的访问：

```
void Shape::set_point(int i, Point p)    // not used; not necessary so far
{
    points[i] = p;
}

Point Shape::point(int i) const
{
    return points[i];
}

int Shape::number_of_points() const
{
    return points.size();
}
```

在派生类的成员函数中，这些函数的使用方法如下：


```

void Lines::draw_lines() const
    // draw lines connecting pairs of points
{
    for (int i=1; i<number_of_points(); i+=2)
        fl_line(point(i-1).x,point(i-1).y,point(i).x,point(i).y);
}

```

你可能会担心那些琐碎的访问函数。它们是不是很低效？会不会使程序变慢？会不会增加程序的大小？不会的，它们都会被编译器以“内联”（inlined）方式进行编译。实际上，调用 `number_of_points()` 跟直接调用 `points.size()` 使用一样多的内存，执行一样多的指令。

这些访问控制的考虑和决定是非常重要的，接近于最小版本的 Shape 类可以定义如下：

```

struct Shape {    // close-to-minimal definition — too simple — not used
    Shape();
    void draw() const;    // deal with color and call draw_lines
    virtual void draw_lines() const;    // draw the appropriate lines
    virtual void move(int dx, int dy);    // move the shape +=dx and +=dy
    vector<Point> points;    // not used by all shapes
    Color lcolor;
    Line_style ls;
    Color fcolor;
};

```

我们增加的 12 个成员函数和两行访问控制说明（`private:` 和 `protected:`）有何价值呢？其基本作用是保护类的描述不会被设计者以不可预见的方式更改，从而使我们能用更少的精力写出更好的类。这就是所谓的“不变式”（参见 9.4.3 节）。下面，我们将通过定义 Shape 类的派生类来说明这一优点。一个简单的例子是 Shape 类的早期版本用到了下面两个成员：

```

Fl_Color lcolor;
int line_style;

```

这种实现方式被证明局限性太大（线型为 `int` 类型不能完美地表示线宽，而 `Fl_Color` 不能表示不可见方式），并且使得代码凌乱。如果这两个变量是公有（`public`）的，并被用户代码所使用，那么改进接口库就只能伴随着重写这些用户代码（因为在用户代码中使用了名字 `line_color` 和 `line_style`）。

另外，访问函数在符号表示方面更为方便。例如，`s.add(p)` 比 `s.points.push_back(p)` 更易读、易写。

14.2.3 绘制形状

我们现在已经介绍了除 Shape 类核心之外的所有内容：

```

void draw() const;    // deal with color and call draw_lines
virtual void draw_lines() const;    // draw the lines appropriately

```

Shape 最基本的功能是绘制形状。但我们不可能将其他所有功能、数据都去掉，而又不给 Shape 造成损害（参见 14.4 节）。绘制是 Shape 的本职工作，它借助 FLTK 和操作系统的基本机制来完成这一工作。但是，从用户的观点来看，它只是提供了以下两个函数：

- `draw()` 函数首先应用线型和颜色设置，然后调用 `draw_lines()`。
- `draw_lines()` 在屏幕上绘制像素。

函数 `draw()` 并没有使用任何新奇的技术，只是简单地调用 FLTK 函数来设置 Shape 中指定的颜色和线型，接着调用 `draw_lines()` 函数在屏幕上进行实际的绘制，最后将颜色和线型恢复到调用之前的情况：

```

void Shape::draw() const
{
    Fl_Color oldc = fl_color();
    // there is no good portable way of retrieving the current style

```

```

    fl_color(lcolor.as_int());           // set color
    fl_line_style(ls.style(),ls.width()); // set style
    draw_lines();
    fl_color(olddc);                     // reset color (to previous)
    fl_line_style(0);                     // reset line style to default
}

```

不幸的是，FLTK 并没有提供获得当前线型的方法，所以线型只是设置为默认值。这就是有些时候为了简单性和可移植性而不得不接受的妥协。我们认为，试图在接口库中实现这一功能是不值得的。

注意，`Shape::draw()` 函数并不处理填充颜色或者线的可见性，这些都由单独的函数 `draw_lines()` 来完成，它对如何解释这些设置有更好的了解。原则上，所有颜色和线型都可以交给 `draw_lines()` 函数来处理，但是重复性会相当高。

现在考虑我们应该如何处理 `draw_lines()` 函数。如果你稍微想一下，就会明白让 `Shape` 类的一个函数来完成多种不同形状的绘制是非常困难的。如果那样做，可能需要在 `Shape` 对象中保存每个形状的最后像素。如果我们继续使用 `vector<Point>` 模型，将会存储非常多的点。更糟糕的是，“屏幕”（即图形硬件）已经做了这些，而且做得更好。

为了避免额外的工作和存储空间，`Shape` 类采用了另外一种方式：它为每种 `Shape`（即每个 `Shape` 的派生类）都提供了定义自己的绘制函数的机会。`Text`、`Rectangle` 或 `Circle` 类都可能有自己的更好的绘制方法。事实上，大部分形状类都是这样。毕竟，这些类确切地“知道”它们所要绘制的内容。例如，将 `Circle` 类定义为一个点和一个半径，远好于许多线段。在需要的时候，通过一个点和一个半径生成要绘制的像素实际上并不像想象的那么困难。因此 `Circle` 类定义了自己的 `draw_lines()` 函数，我们更希望调用这个函数而不是 `Shape` 类的 `draw_lines()` 函数。这就是将 `Shape::draw_lines()` 声明为 `virtual` 的意义所在：

```

struct Shape {
    // ...
    virtual void draw_lines() const; // let each derived class define its
                                    // own draw_lines() if it so chooses
    // ...
};

struct Circle : Shape {
    // ...
    void draw_lines() const; // "override" Shape::draw_lines()
    // ...
};

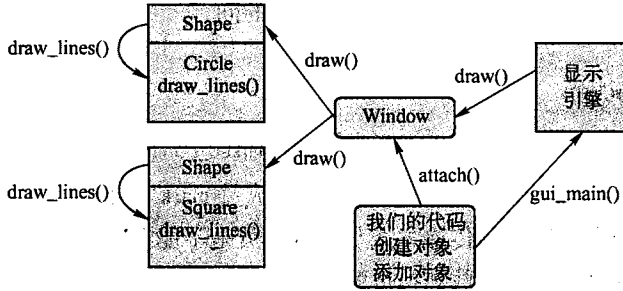
```

所以，如果 `Shape` 是一个 `Circle` 对象，`Shape` 的 `draw_lines()` 就会以某种方式调用 `Circle` 的一个版本；同样，如果是一个 `Rectangle` 对象，就会调用 `Rectangle` 的一个版本。这正是 `draw_lines()` 声明中关键字 `virtual` 的含义：如果一个派生自 `Shape` 的类定义了自己的 `draw_lines()` 函数（和 `Shape` 类的 `draw_lines()` 函数有相同的类型），那么此 `draw_lines()` 将被调用，而不是 `Shape` 类的 `draw_lines()`。第 13 章显示了这一机制是如何在 `Text`、`Circle`、`Closed_polyline` 等类中起作用的。在派生类中定义一个函数，使之可以通过基类提供的接口进行调用，这种技术称为覆盖（overriding）。

注意，尽管在 `Shape` 类中处于核心地位，`draw_lines()` 还是被定义成 `protected`，这意味着它不能被“一般用户”调用——这是 `draw()` 的目的而不是 `draw_lines()` 的，`draw_lines()` 只是作为一个“实现细节”被 `draw()` 函数以及 `Shape` 的派生类使用。

这样就完成了 12.2 节中的显示模型。驱动屏幕的系统了解 `Window` 类，`Window` 类了解 `Shape`

类并可以调用它的 `draw()` 函数。最后, `draw()` 函数调用特定形状类的 `draw_lines()` 函数。我们的用户代码对 `gui_main()` 函数的调用会启动这个显示引擎, 如下图所示:



`gui_main()` 函数是什么? 到目前为止, 我们还没有在代码中实际看到过它。作为替代, 我们使用了 `wait_for_button()`, 它用更单纯的方式调用显示引擎。

`Shape` 类的 `move()` 函数简单地将保存的每个点相对于当前位置移动一个偏移量:

```
void Shape::move(int dx, int dy)    // move the shape +=dx and +=dy
{
    for (int i = 0; i < points.size(); ++i) {
        points[i].x += dx;
        points[i].y += dy;
    }
}
```

像 `draw_lines()` 一样, `move()` 也是虚函数, 因为派生类可能包含所要移动的数据, 而 `Shape` 并不知道。例如, 参考 `Axis` (参见 12.7.3 节和 15.4 节)。

逻辑上, `move()` 函数对于 `Shape` 类并不是必须的, 提供它只是为了方便, 同时也是为了提供另一个虚函数的例子。只要形状类包含了不在 `Shape` 类中存储的点, 就应该定义自己的 `move()` 函数。

14.2.4 拷贝和可变性

`Shape` 类将拷贝构造函数和拷贝赋值运算符声明为 `private`:

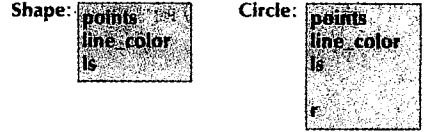
```
private:
    Shape(const Shape&);           // prevent copying
    Shape& operator=(const Shape&);
```

这样做的效果是只有 `Shape` 的成员可以使用默认的拷贝操作来拷贝 `Shape` 对象。这是为了防止意外拷贝而经常使用的一种方法。例如:

```
void my_fct(const Open_polyline& op, const Circle& c)
{
    Open_polyline op2 = op;    // error: Shape's copy constructor is private
    vector<Shape> v;
    v.push_back(c);            // error: Shape's copy constructor is private
    // ...
    op = op2;                  // error: Shape's assignment is private
}
```

但是拷贝在很多地方都非常有用! 你只要看一下 `push_back()` 函数, 没用拷贝功能, 我们甚至无法使用 `vector` (`push_back()` 将参数的一份拷贝放在向量中)。为什么防止拷贝会给程序员带来麻烦呢? 对一个类型而言, 如果默认的拷贝操作可能引起麻烦, 你可以将其禁止。关于“麻烦”的一个很好的例子是 `my_fct()`, 由于 `v` 中的元素“槽”的大小为一个 `Shape`, 所以我们不能将一个 `Circle` 对象拷贝到其中。 `Circle` 对象包含半径数据而 `Shape` 对象没有, 所以 `sizeof(Shape) < sizeof(Circle)`。如果允许执行 `v.push_back(c)`, 则这个 `Circle` 对象将被“切断”存入 `v` 的 `Shape` 元素中, 之后

任何对此 Shape 元素的使用都可能会引起崩溃, 因为对 Circle 的操作都假定它包含一个表示半径的成员(r), 而它并没有拷贝过来, 如右图所示。op2 的拷贝构造函数和向 op 的赋值操作也面临着完全一样的问题。考虑如下情况:



```
Marked_polyline mp("x");
Circle c(p,10);
my_fct(mp,c); // the Open_polyline argument refers to a Marked_polyline
```

现在 Open_polyline 的拷贝操作会将对象 mp 的 string 成员 mark“切断”。

基本上, 类层次结合参数引用传递方式与默认拷贝是不能混合的。当你设计一个将要作为基类的类时, 应禁用它的拷贝构造函数和拷贝赋值操作, 就像我们对 Shape 所做的那样。

切断(是的, 这确实是一个技术术语)并不是我们禁止拷贝的唯一原因。如果没有拷贝操作, 则很多思想可以更好地实现。回忆一下, 图形系统不得不记住 Shape 对象的存储位置, 以便将它显示在屏幕上。这就是为什么我们要将 Shape“添加”(attach)而不是拷贝到 Window。例如, 如果窗口只保存了 Shape 的一个副本, 而不是引用, 那么, 对原 Shape 对象的任何修改都不会影响到副本。那么, 如果我们改变形状的颜色, 窗口将不会知道这个变化, 仍会用原来的颜色来显示副本。因此, 拷贝一个副本在实际中并不如使用原始版本好。

如果我们希望拷贝不同类型的对象, 而默认拷贝操作被禁用了, 可以实现一个显式函数来完成这个工作。这种拷贝函数通常称为 clone()。很显然, 只有当成员读取函数能充分表达构造副本需要什么内容时, 我们才能编写出 clone() 函数, 而所有的形状类恰好都是这种情况。

14.3 基类和派生类

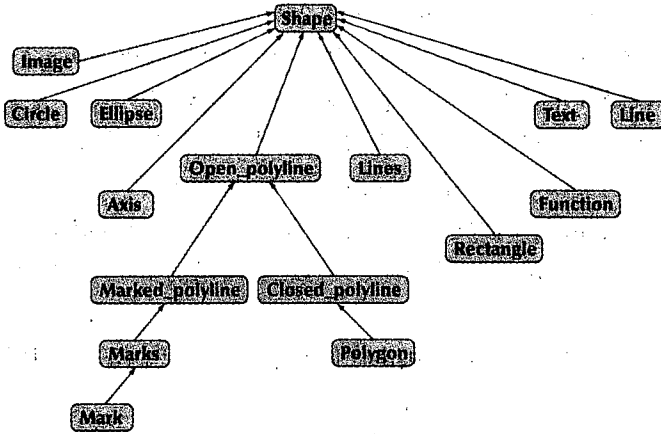
让我们从一个更为技术性的角度来观察基类和派生类, 也就是说, 在本节中(只在本节)我们将讨论的焦点从程序设计、应用设计和图形转移到程序设计语言的特性上来。当设计一个图形接口库时, 我们依赖以下 3 个关键的语言机制:

- 派生(derivation): 从一个类构造另一个类的方法, 使新构造的类可以替换原来的类。例如, Circle 类派生自 Shape 类, 或者换句话说, “Circle 是某种 Shape”或者“Shape 是 Circle 的基类”。派生类(这里是 Circle)除了自己的成员以外, 还包括基类(这里是 Shape)的所有成员。这通常称为继承(inheritance), 因为派生类“继承”了其基类的所有成员。在某些上下文环境中, 派生类称为子类(subclass), 而基类称为父类(Superclass)。
- 虚函数(virtual function): 在基类中定义一个函数, 在派生类中有一个类型和名称完全一样的函数, 当用户调用基类函数时, 实际调用的是派生类中的函数。例如, 当 Window 对 Circle(添加到 Window 的 Shape)调用 draw_lines() 函数时, Circle 类的 draw_lines() 函数得到执行, 而不是 Shape 类本身的 draw_lines() 函数。这通常称为运行时多态(run-time polymorphism)、动态分派(dynamic dispatch)或运行时分派(run-time dispatch), 因为具体调用哪个函数是根据运行时实际使用的对象类型来确定的。
- 私有和保护成员(Private and protected member): 我们保持类的实现细节为私有的, 以保护它们不被直接访问, 简化维护操作, 这通常称为封装(encapsulation)。

继承、运行时多态和封装的使用, 实际上就是面向对象程序设计(object-oriented programming)最常见的标志。因此, 除了其他的程序设计风格之外, C++ 还直接支持面向对象程序设计。例如, 在第 20 和 21 章中, 我们将看到 C++ 如何支持泛型编程。C++ 借用了 Simu-

la67 语言(给予了明确的致谢)的核心机制, Simula67 是第一个直接支持面向对象程序设计的语言(参见第 22 章)。

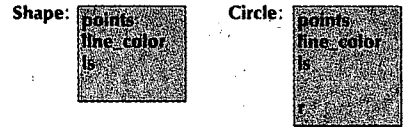
这里有很多技术术语!但是它们代表什么意思?同时它们在计算机中实际是如何工作的?我们首先为图形接口类画一个简单的继承关系图:



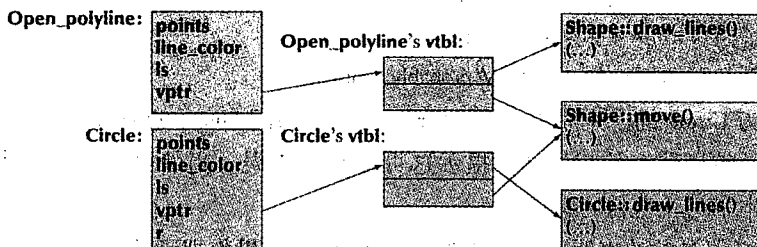
箭头从派生类指向它的基类。这种图示可以帮助我们看到类之间的关系,因此经常会出现程序员的黑板上。与一个商业框架相比,这是一个非常小的“类层次”,仅仅包含 16 个类,而且只有 Open_polyline 类的后代才会有多于一层的情况。很明显,虽然代表的是一个抽象概念,我们永远不能直接创建其对象,公共基类(Shape)仍是最重要的一个类。

14.3.1 对象布局

对象在内存中是如何布局的呢?就像我们在 9.4.1 节中所看到的,一个类的成员定义了对象的布局:数据成员在内存中一个接一个地存储。当使用继承时,派生类的数据成员被简单地放在基类的成员之后,如右图所示。一个 Circle 对象包含 Shape 类的数据成员(毕竟,它也是一种 Shape),并且可以当做 Shape 对象使用。此外, Circle 对象还有它自己的数据成员 r,存放在继承的数据成员之后。



为了处理一个虚函数调用,我们需要(并且必须)在 Shape 对象中存储更多的信息:当我们调用 Shape 的 draw_lines() 函数时,可以借助这些信息分辨出实际应该调用哪个函数。常用的方法是增加一个函数列表的地址,这个表通常称为 vtbl(即“virtual table”或“virtual function table”,虚函数表),它的地址通常称为 vptr(即“virtual pointer”,虚指针)。我们将在第 17~18 章中讨论指针,在这里,它们的作用类似于引用。对于 vtbl 和 vptr,一个给定的实现可能使用不同的名字。将 vptr 和 vtbl 加入布局图中可得到下图:



因为 draw_lines() 函数是第一个虚函数,所以它占据了 vtbl 中的第一个位置,紧接着是第二个虚

函数 `move()`。只要你需要，一个类可以有任意多个虚函数，其 `vtbl` 的规模则视需要而定（一个位置对应一个虚函数）。当我们调用 `x.draw_lines()` 的时候，编译器查找 `x` 的 `vtbl` 中 `draw_lines()` 的对应位置，调用找到的函数。基本上，代码只不过是按照图中箭头寻找对应的函数而已。因此，如果 `x` 是一个 `Circle`，`Circle::draw_lines()` 将会被调用。如果 `x` 是另一个类型，比如 `Open_polyline`，而它的 `vtbl` 与 `Shape` 类一样，则 `Shape::draw_lines()` 将会被调用。类似地，由于 `Circle` 没有定义它自己的 `move()` 函数，所以如果 `x` 是一个 `Circle`，则 `x.move()` 将调用 `Shape::move()`。基本上，虚函数调用产生的目标代码首先简单地寻找 `vptra`，通过它找到对应的 `vtbl`，然后调用其中正确的函数。其代价大约是两次内存访问加上一次普通函数调用，既简单又快速。

`Shape` 是一个抽象类，所以我们不能实际拥有一个 `Shape` 对象。但是一个 `Open_polyline` 对象拥有和“平凡形状”一样的布局，因为它并没有增加数据成员，也没有定义虚函数。对于每个具有虚函数的类，只有一个全局的 `vtbl`，而不是每个对象都有自己的 `vtbl`，所以 `vtbl` 并不会明显地增加程序目标代码的大小。

注意，在上面的布局图中，我们没有画出任何非虚函数。我们不需要那样做，因为那些函数的调用方式没有任何特别之处，所以它们不会增加对象的大小。

定义一个和基类中虚函数的名称和类型都相同的函数（比如 `Circle::draw_lines()`），以使派生类的函数代替基类中的版本被放入 `vtbl` 中的技术称为覆盖。例如，`Circle::draw_lines()` 覆盖了 `Shape::draw_lines()`。

我们为什么要告诉你这些关于 `vtbl` 和内存布局的内容呢？为了进行面向对象程序设计，你需要了解这些内容吗？实际上并不需要，但很多人非常想知道事情是如何实现的（我们也是如此），而当人们不理解事情的时候，荒诞的说法就会产生。我们遇到过一些讨厌虚函数的人，“因为它们代价很高”。为什么？如何得出代价高的结论？和谁相比？什么情况下这些代价会产生问题？我们解释了虚函数的实现模型后，你就不会再有这些恐惧了。如果你需要一个虚函数调用（在运行时选择被调用函数），你不可能使用其他语言特性编写出速度更快或者使用更少内存的代码。这一点很容易理解。

14.3.2 类的派生和虚函数定义

我们通过在类名后给出一个基类来指定一个类为派生类。例如：

```
struct Circle : Shape { /* ... */};
```

默认情况下，结构体（`struct`）的成员都是公有的（参见 9.3 节），基类中的公有成员也会成为结构体的公有成员。另一种等价的定义方式如下：

```
class Circle : public Shape { public: /* ... */};
```

这两种 `Circle` 的声明是完全等价的，至于哪一种方式更好，可能你和其他人争论很长时间也没有结论。我们的意见是，不如把时间花在其他问题上，可能更有价值。

注意不要忘记了 `public` 关键字。例如：

```
class Circle : Shape { public: /* ... */}; // probably a mistake
```

这将使 `Shape` 成为 `Circle` 的一个私有基类，`Circle` 将不能访问 `Shape` 的公有函数。这很可能不是你想要的，一个好的编译器会给出警告，提示这可能是个错误。当然也有私有基类正确使用的例子，不过那不在本书讨论范围之内。

一个虚函数必须在类的声明中被声明为 `virtual`，但是如果你把函数定义放在类外，关键字 `virtual` 就不必也不能出现在那里了。例如：


```

struct Shape {
    // ...
    virtual void draw_lines() const;
    virtual void move();
    // ...
};

virtual void Shape::draw_lines() const { /* ... */ } // error
void Shape::move() { /* ... */ } // OK

```

14.3.3 覆盖

当你希望覆盖一个虚函数时，必须使用与基类中完全相同的名字和类型。例如：

```

struct Circle : Shape {
    void draw_lines(int) const; // probably a mistake (int argument?)
    void drawlines() const;    // probably a mistake (misspelled name?)
    void draw_lines();        // probably a mistake (const missing?)
    // ...
};

```

这里，编译器会看到 3 个与 `Shape::draw_lines()` 无关的函数（因为它们有不同的名字或者不同的类型），这些函数没有覆盖它。一个好的编译器会给出警告，提示这些可能是错误。你不能也不必在覆盖函数中加入一些内容来保证它确实覆盖了一个基类的函数。

`draw_lines()` 是一个真实的例子，难以模仿其所有细节来学习覆盖技术。因此，我们下面给出一个纯技术性的例子来说明覆盖：

```

struct B {
    virtual void f() const { cout << "B::f "; }
    void g() const { cout << "B::g "; } // not virtual
};

struct D : B {
    void f() const { cout << "D::f "; } // overrides B::f
    void g() { cout << "D::g "; }
};

struct DD : D {
    void f() { cout << "DD::f "; } // doesn't override D::f (not const)
    void g() const { cout << "DD::g "; }
};

```

这段代码给出了一个简单的类层次关系，仅仅包含一个虚函数 `f()`。我们可以试着使用它。特别地，我们可以试着调用 `f()` 和非虚函数 `g()`。除非要处理的对象的类型是 `B`（或者是 `B` 的派生类），否则 `g()` 并不知道类型是什么：

```

void call(const B& b)
    // a D is a kind of B, so call() can accept a D
    // a DD is a kind of D and a D is a kind of B, so call() can accept a DD
{
    b.f();
    b.g();
}

int main()
{
    B b;
    D d;
    DD dd;

    call(b);
}

```

```

    call(d);
    call(dd);

    b.f();
    b.g();

    d.f();
    d.g();

    dd.f();
    dd.g();
}

```

你将得到

```
B::f B::g D::f B::g D::f B::g B::f B::g D::f D::g DD::f DD::g
```

当你理解了为什么是这样的输出结果以后，你就会明白继承和虚函数机制了。

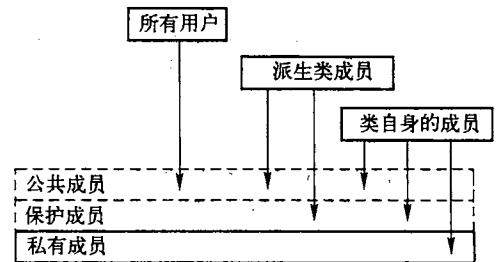
14.3.4 访问

C++ 为类成员访问提供了一个简单的模型。类的成员可以是：

- 私有的 (private)：如果一个成员是私有的，它的名字只能被其所属类的成员使用。
- 受保护的 (protected)：如果一个成员是受保护的，它的名字只能被其所属类及其派生类的成员使用。
- 公有的 (public)：如果一个成员是公有的，它的名字可以被所有函数使用。

这一模型也可以图形化表示见右图。基类可以是私有的、受保护的或公有的：

- 如果类 D 的一个基类是 private 的，它的 public 和 protected 成员的名字只能被类 D 的成员使用。
- 如果类 D 的一个基类是 protected 的，它的 public 和 protected 成员的名字只能被类 D 及其派生类的成员使用。
- 如果类的一个基类是 public 的，它的名字可以被所有函数使用。



这些定义忽略了“友元”(friend)的概念和一些次要的细节，那不在本书讨论范围之内。如果你想成为语言专家，你需要学习 Stroustrup 的《The Design and Evolution of C++》、《The C++ Programming Language》和《2003 ISO C++ 标准》。但我们并不推荐你成为语言专家(知道语言定义的每一个微小细节)，作为一名程序员(一位软件开发者、工程师、用户以及所有实际使用语言的人)乐趣更多，对社会也更有价值。

14.3.5 纯虚函数

一个抽象类是一个只能作为基类的类。我们使用抽象类来表示那些抽象的概念，即相关实体共性的一般化所对应的那些概念。人们曾写过很厚的哲学书试图精确地定义抽象概念(或抽象或一般性等)。无论其哲学定义如何，抽象概念的思想是极其有用的。例如，“动物”(相对于任何特定种类动物)、设备驱动程序(相对于某种特定设备的驱动程序)和出版物(相对于任何特定种类的书或杂志)。在程序中，抽象类通常定义了一组相关类(类层次)的接口。

在 14.2.1 节中，我们看到了如何通过声明 protected 构造函数来定义一个抽象类。下面是另一种更常用的方法：声明一个或者多个必须在派生类中被覆盖的虚函数。例如：


```

class B { // abstract base class
public:
    virtual void f()=0; // pure virtual function
    virtual void g()=0;
};

B b; // error: B is abstract

```

这个奇怪的语法 `=0` 指出 `B::f()` 和 `B::g()` 是“纯”虚函数，即它们必须在派生类中被覆盖。因为 `B` 有纯虚函数，所以我们不能创建一个 `B` 的对象。覆盖纯虚函数可解决这个“问题”：

```

class D1 : public B {
public:
    void f();
    void g();
};

D1 d1; //ok

```

注意，除非所有纯虚函数都被覆盖了，否则该派生类也是抽象的：

```

class D2 : public B {
public:
    void f();
    // no g()
};

D2 d2; // error: D2 is (still) abstract
class D3 : public D2 {
public:
    void g();
};

D3 d3; //ok

```

通常，带有纯虚函数的类的目标是提供纯粹的接口，即它们倾向于不包含任何数据成员（数据成员在派生类中定义），因此没用任何构造函数（如果没有任何数据成员需要初始化，那么就不需要构造函数）。

14.4 面向对象程序设计的好处

当我们说 `Circle` 派生自 `Shape`，或者 `Circle` 是一种 `Shape` 的时候，实际上获得了如下好处（其中之一或者两者皆有）：

- 接口继承 (interface inheritance)：需要 `Shape` 对象参数（通常作为一个引用参数）的函数可以接受 `Circle` 对象参数（并且可以通过 `Shape` 提供的接口使用 `Circle`）。
- 实现继承 (implementation inheritance)：当定义 `Circle` 及其成员函数时，我们可以使用 `Shape` 提供的功能（如数据成员和成员函数）。

一个不能提供接口继承的设计（即一个派生类的对象不能当做其公有基类的对象使用）是一个拙劣且容易出错的设计。例如，我们可能定义一个以 `Shape` 为公有基类的类 `Never_do_this`，然后定义函数覆盖 `Shape::draw_lines()`，它不绘制任何形状，相反，将自己的中心向左移动 100 个像素。这个“设计”是有致命缺陷的，因为尽管 `Never_do_this` 提供了一个 `Shape` 的接口，但其实现没有保持 `Shape` 所要求的语义（意义、行为）。永远不要这样做！

接口继承之所以得名，是因为其优点：代码使用基类（“接口”，这里是 `Shape`）提供的接口，而无需知道具体的派生类（“实现”；这里是 `Shape` 的派生类）。

实现继承之所以得名，是因为其优点：通过使用基类提供的功能，简化了派生类的实现。

注意，我们的图形库设计严重依赖接口继承：“图形引擎”调用 `Shape::draw()`，接着 `Shape::draw()` 将调用 `Shape` 的虚函数 `draw_lines()` 完成实际的图形显示工作。无论“图形引擎”还是实际 `Shape` 类都不知道有哪些具体形状。特别是，“图形引擎”(FLTK 加上操作系统的图形功能)是在设计图形类之前若干年就编写、编译好的！它根本无法知道图形类的任何信息。我们只是定义了特定的形状并且将它们当做 `Shape` 对象添加到了 `Window` 中(`Window::attach()` 接受一个 `Shape&` 类型的参数；参见附录 E.3)。而且，由于 `Shape` 类不知道你的图形类，当你每次定义一个新的图形接口类时，不需要重新编译 `Shape` 类。

换句话说，我们可以向程序中加入新形状，而不用修改已有的代码。这是一个软件设计/开发/维护的圣杯：扩展一个系统而不用修改它。哪些改进不必修改已有类还是有一定限制的(例如，`Shape` 提供了非常有限的服务)，同时这种技术也不是对所有的程序设计问题都能很好地应用(例如，第 17~19 章定义的 `vector`，继承机制对其没什么用处)。然而无论如何，接口继承是设计和实现对于改进需求鲁棒性很强的系统的最有力的技术之一。

同样，实现继承也能带来很多好处，但是世界上没有万能灵药。通过在 `Shape` 中放入有用的服务，我们避免了在派生类中一遍又一遍地进行重复性工作的烦恼。这对现实世界中的程序设计尤为重要。然而，它带来了一个额外代价，任何对于 `Shape` 接口或者对于 `Shape` 数据成员布局的更改都必须重新编译所有的派生类及其用户代码。对于一个广泛使用的库来说，这种重新编译是绝对行不通的。当然，有一些方法可以在得到大多数好处的同时避免大多数的问题，参见 14.3.5 节。

简单练习

不幸的是，我们无法构造一个能帮助理解一般设计原则的简单练习，所以在本练习中我们把注意力集中在支持面向对象程序设计的语言特性上。

1. 定义带有一个虚函数 `vf()` 和一个非虚函数 `f()` 的类 `B1`。在 `B1` 内定义这两个函数，使它们都输出自己的名字(例如“`B1::vf()`”)。将这两个函数定义为公有的。建立一个 `B1` 对象并且调用每个函数。
2. 从 `B1` 类派生一个 `D1` 类，并且覆盖 `vf()`。建立一个 `D1` 对象，并调用 `vf()` 和 `f()`。
3. 定义一个 `B1` 的引用(`B1&`)并且初始化为一个 `D1` 对象，并调用 `vf()` 和 `f()`。
4. 为 `D1` 定义一个 `f()` 函数，重做练习 1~3，并解释其结果。
5. 在 `B1` 中定义一个纯虚函数 `pvf()`，重做练习 1~4，并解释其结果。
6. 定义一个派生自 `D1` 的 `D2` 类，并且在 `D2` 中覆盖 `pvf()`。建立一个 `D2` 类的对象并且调用 `f()`、`vf()`、`pvf()` 函数。
7. 定义带有一个纯虚函数 `pvf()` 的 `B2` 类。定义 `D21` 类，包含一个 `string` 数据成员和一个覆盖 `pvf()` 的成员函数，`D21::pvf()` 输出 `string` 数据成员的值。定义 `D22` 类，它与 `D21` 类一样，只是数据成员为 `int` 类型。定义函数 `f()`，接受一个 `B2&` 参数，并对此参数调用 `pvf()` 函数。使用 `D21` 对象和 `D22` 对象调用 `f()`。

思考题

1. 什么是应用领域？
2. 什么是理想的命名？
3. 我们可以命名哪些东西？
4. `Shape` 类提供了哪些功能？
5. 如何区别抽象类和非抽象类？
6. 如何将类设计为抽象类？
7. 访问控制能够控制什么？

8. 私有(private)数据成员有什么好处?
9. 虚函数是什么? 如何区别于一个非虚函数?
10. 什么是基类?
11. 如何定义一个派生类?
12. 对象的布局意味着什么?
13. 使一个类更易于测试, 应该做哪些工作?
14. 继承关系图是什么?
15. 保护(protected)对象和私有(private)对象有什么区别?
16. 类中的哪些成员可以被它的派生类访问?
17. 如何区别纯虚函数和其他虚函数?
18. 为什么将一个成员函数设计为虚函数?
19. 为什么将一个虚函数设计为纯虚函数?
20. 覆盖的含义是什么?
21. 接口继承和实现继承有什么区别?
22. 什么是面向对象程序设计?



术语

抽象类	可变性	纯虚函数	访问控制	对象布局	子类
基类	面向对象	超类	派生类	多态	虚函数
分派	私有	虚函数调用	封装	保护	虚函数表
继承	公有				



习题

1. 定义两个类 Smiley 和 Frowny, 它们都派生自 Circle 类, 并且有两只眼睛和一张嘴。接下来, 分别从 Smiley 和 Frowny 类派生类, 为其添加一个适当的帽子。
2. 尝试拷贝一个 Shape 对象, 会发生什么?
3. 定义一个抽象类并且尝试定义一个该类型的对象, 会发生什么?
4. 定义一个类似于 Circle 的 Immobile_Circle 类, 只是它不能移动。
5. 定义 Striped_rectangle 类, 不采用标准的填充方式, 而是用一个像素宽的水平线“填充”该矩形的内部(比如每隔一个像素画一条线)。你可能需要通过设置线宽和线间距来获得喜欢的图案。
6. 使用 Striped_rectangle 中的技术定义 Striped_Circle 类。
7. 使用 Striped_rectangle 中的技术定义 Striped_closed_polyline 类(需要一些算法上的创新)。
8. 定义 Octagon 类, 表示正八边形。编写测试程序, 测试它的所有成员函数(包括你自己定义的和继承自 Shape 类的)。
9. 定义 Group 类, 表示 Shape 的容器, 为其设计适合的操作, 能恰当处理类的不同成员。提示: 使用 Vector_ref。利用 Group 定义一个国际跳棋棋盘, 棋子可以在程序的控制下移动。
10. 定义一个非常像 Window 的 Pseudo_window 类(尽你所能, 但不必花费太大精力)。它应该是圆角的, 应带有标签和控制图标。也许你可以添加一些假的“内容”, 如一幅图像。它不必做任何实质性工作。一种可接受的方法(实际上我们建议这样做)是将其显示在一个 Simple_window 中。
11. 定义一个 Binary_tree 类, 它派生自 Shape 类。层数作为一个参数(levels == 0 表示没有节点, levels == 1 表示有一个节点, levels == 2 表示有一个顶层节点和两个子节点, levels == 3 表示有一个顶层节点、两个子节点以及这两个子节点的各自两个子节点, 依此类推)。使用小圆圈表示一个节点, 并用线连接这些节点。注意: 在计算机科学中, 树是从一个顶层节点(有趣且合乎逻辑的是它经常被称为根)向下生长的。
12. 修改 Binary_tree, 使用虚函数来绘制它的节点。然后, 从 Binary_tree 派生一个新类, 对节点使用一个不同的表示(比如, 一个三角形)来覆盖此虚函数。

13. 修改 `Binary_tree`, 使其接受一个参数(或者多个)来指出用什么类型的线连接这些节点(例如, 一个向下箭头或者一个红色的向下箭头)。注意, 本题和习题 12 是如何使用两种不同的方式使得类的层次结构更加灵活和有用的。
14. 为 `Binary_tree` 类增加一个操作, 将文本添加到节点上。你可能必须修改 `Binary_tree` 的设计来实现这个功能。选择一种方式来标识节点, 例如, 你可以用字符串“lrrl”表示向下遍历二叉树的左、右、右、左和右会到达当前节点(以 l 或者 r 开头都可与根节点匹配)。
15. 大多数类层次是与图形无关的。定义 `Iterator` 类, 它包含一个返回值为 `double *` 类型的纯虚函数 `next()`。基于 `Iterator` 类派生 `Vector_iterator` 和 `List_iterator` 类, 使 `Vector_iterator` 的 `next()` 函数生成指向 `vector < double >` 中下一个元素的指针, 而 `List_iterator` 对于 `list < double >` 类型实现相同的操作。`Vector_iterator` 对象通过一个 `vector < double >` 初始化, 对于 `next()` 的第一次调用应得到指向第一个元素的指针(如果向量不为空的话)。如果没有下一个元素的话, `next()` 应返回 0。编写函数 `void print(Iterator&)`, 打印 `vector < double >` 和 `list < double >` 中的元素, 从而实现测试。
16. 定义 `Controller` 类, 它包含 4 个虚函数 `on()`、`off()`、`set_level(int)` 和 `show()`。至少从 `Controller` 派生出两个类, 第一个派生类是一个简单的测试类, 它的 `show()` 函数打印出这个类是设置为开还是关以及当前的级别; 第二个派生类需要以某种方法控制一个 `Shape` 对象的线的颜色, “级别”的确切含义由你自己决定。试着找到 `Controller` 类可以控制的第三种“东西”。
17. 在 C++ 标准库中定义的异常, 例如 `exception`、`runtime_exception` 和 `out_of_range`(参见 5.6.3 节), 被组织为一个类层次(使用一个很有用的虚函数 `what()`, 它返回一个字符串来解释发生了什么错误)。查找 C++ 标准异常类的层次结构, 绘制它的类层次图。



附言

软件设计的理想不是构造一个可以做任何事情的程序, 而是构造很多类, 这些类可以准确反应我们的思想, 可以组合在一起工作, 允许我们用来构造漂亮的应用程序, 并且具有最小的工作量(相对于任务的复杂度而言)、足够高的性能以及保证产生正确的结果等优点。这样的程序易于理解、易于维护, 而简单地将一些代码快速拼凑在一起来完成某个特定工作则不可能具有这样的优点。类、封装(由 `private` 和 `protected` 支持)、继承(由类的派生支持)和运行时多态(由虚函数支持)都是我们构建系统的最有力的工具。

第 15 章 绘制函数图和数据图

“至善者善之敌。”

——Voltaire

如果是从事实验领域，你需要用图表示数据；如果是从事自然现象的数学建模领域，你需要用图表示函数。本章将讨论这类图形的基本机制。像往常一样，我们将介绍这些机制的使用方法以及它们的设计理念。本章给出的关键实例是绘制一个带有单参数的函数图，以及显示从文件中读取的值。

15.1 介绍

与可视化领域的专业软件相比，本章介绍的工具比较原始。我们的主要目标不是输出的美观性，而是理解如何生成这样的图形输出以及其中所使用的编程技术。你会发现，本章使用的设计方法、编程技术和基本数学工具比提出的图形工具有更长久的价值。因此，请不要快速掠过那些代码段，代码比它们计算和绘制出来的形状更重要。

15.2 绘制简单函数图

让我们开始吧。首先看一些例子，它们展示了我们能绘制什么图形以及使用什么样的代码来绘制。特别地，请仔细阅读所使用的图形接口类。此处，我们首先绘制了一条抛物线、一条水平线和一条斜线，如右图所示。实际上，因为本章介绍函数的图形化，所以这条水平线并不仅仅是一条水平线，它是我们将此函数图形化得来的：

```
double one(double) { return 1; }
```

这大概是我们能想到的最简单的函数：该函数只有一个参数，而且对任何参数都返回 1。因为我们不需要用这个参数来计算结果，所以我们不需要为它命名。对于每一个传递给 `one()` 函数的参数 x ，我们得到 y 的值都是 1；也就是说，对所有的 x 而言，这条线由 $(x, y) == (x, 1)$ 定义。

像所有初级的数学参数一样，本例有些过于简单，所以让我们看一个稍微复杂一些的函数：

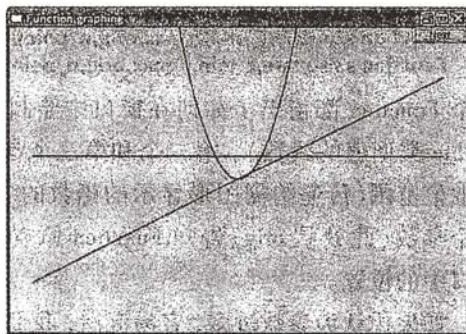
```
double slope(double x) { return x/2; }
```

这是生成斜线的函数。对每一个 x ，我们得到的 y 值为 $x/2$ 。换句话说， $(x, y) == (x, x/2)$ 。两条线的交点是 $(2, 1)$ 。

现在我们可以试验一些更有趣的函数，平方函数似乎是有规律地在本书中重复出现：

```
double square(double x) { return x*x; }
```

如果你还记得中学的几何课程（即使不记得了也不要紧），这个函数定义了一个最低点在 $(0, 0)$ 且以 y 轴对称的抛物线。换句话说， $(x, y) == (x, x * x)$ 。所以，抛物线的最低点和斜线相交于点 $(0, 0)$ 。



下面是绘制这三个函数的代码:

```
const int xmax = 600;      // window size
const int ymax = 400;

const int x_orig = xmax/2; // position of (0,0) is center of window
const int y_orig = ymax/2;
const Point orig(x_orig,y_orig);

const int r_min = -10;     // range [-10:11)
const int r_max = 11;

const int n_points = 400;  // number of points used in range

const int x_scale = 30;    // scaling factors
const int y_scale = 30;

Simple_window win(Point(100,100),xmax,ymax,"Function graphing");
Function s(one,r_min,r_max,orig,n_points,x_scale,y_scale);
Function s2(slope,r_min,r_max,orig,n_points,x_scale,y_scale);
Function s3(square,r_min,r_max,orig,n_points,x_scale,y_scale);

win.attach(s);
win.attach(s2);
win.attach(s3);
win.wait_for_button();
```

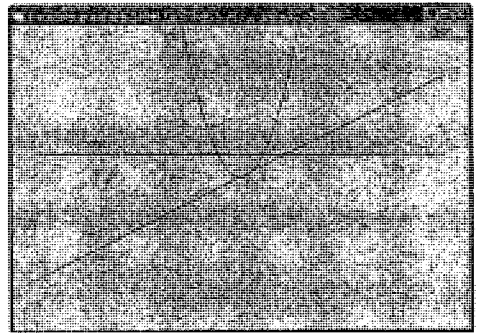
首先,我们定义了一组常量,这样就不必用“魔数”来弄乱我们的代码了。然后,我们创建了一个窗口,定义这些函数,将它们添加到窗口上,最后将控制权交给图形系统进行实际的绘制。

除了三个 Function s、s2、s3 的定义外,其余的都是重复性的和“样板”代码:

```
Function s(one,r_min,r_max,orig,n_points,x_scale,y_scale);
Function s2(slope,r_min,r_max,orig,n_points,x_scale,y_scale);
Function s3(square,r_min,r_max,orig,n_points,x_scale,y_scale);
```

每个 Function 都指出了如何在窗口中绘制其第一个参数(接受一个 double 类型参数并返回一个 double 类型值的函数),第二个和第三个参数给出 x 的取值范围(传递给需图形显示的函数的参数),第四个参数(此处是 orig)告知 Function 原点(0, 0)在窗口中的位置。

如果你认为参数过多,容易混淆,我们同意。我们的理想方案是使用尽可能少的参数,因为参数过多会造成混淆且容易出错。但是,本例确实需要这么多参数。我们将在后面(15.3 节)解释后三个参数。无论如何,我们先为图形添加标签,如右图所示。

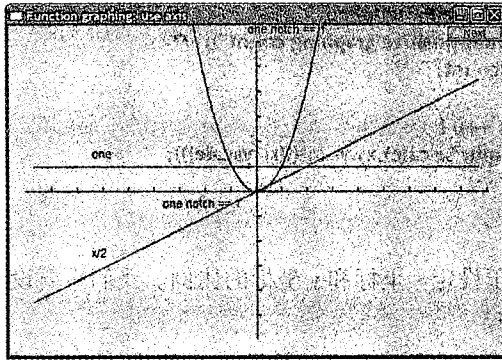


如果我们总是尝试使图形能自我解释。人们不会总是去阅读图形周围的解释文字,而且图像可能会被移动,从而导致解释文字“丢失”。我们放在图片中的任何内容都会成为图片的一部分,更容易被读者注意到。如果是合理的内容,还能帮助读者理解我们所显示的图形。此处,我们简单地给每个图形添加了一个标签。“添加标签”的代码使用了三个 Text 对象(参见 13.11 节):

```
Text ts(Point(100,y_orig-40),"one");
Text ts2(Point(100,y_orig+y_orig/2-20),"x/2");
Text ts3(Point(x_orig-100,20),"x*x");
win.set_label("Function graphing: label functions");
win.wait_for_button();
```

从现在开始，我们将省略掉一些重复的代码，包括将形状添加到窗口、为窗口添加标签和等待用户点击“Next”按钮等代码。

但是，这样的图片仍然是不可接受的。我们注意到 $x/2$ 与 $x * x$ 相交于点 $(0, 0)$ ，同时 one 与 $x/2$ 相交于点 $(2, 1)$ ，但这些现象有些难以察觉；我们需要使用坐标轴来清楚地展现给读者：



实现坐标轴的代码使用了两个 Axis 对象(参见 15.4 节)：

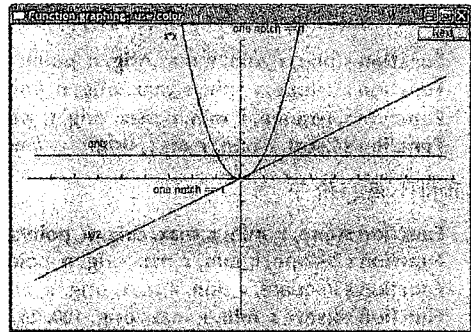
```
const int xlength = xmax-40; // make the axis a bit smaller than the window
const int ylength = ymax-40;

Axis x(Axis::x, Point(20, y_orig), xlength, xlength/x_scale, "one notch == 1");
Axis y(Axis::y, Point(x_orig, ylength+20),
    ylength, ylength/y_scale, "one notch == 1");
```

刻度的数量为 $xlength/x_scale$ ，这样每个刻度分别代表数值 1、2、3 等。按照惯例，坐标轴相交于点 $(0, 0)$ 。如果你更愿意，当然也可以让它们分别沿着窗口左侧和底部的边缘，就像显示数据的惯例那样(参见 15.6 节)。另一种区别坐标轴和数据的方法是使用不同颜色：

```
x.set_color(Color::red);
y.set_color(Color::red);
```

于是可得到右图。这是一个可以接受的输出结果了，虽然出于美观的原因，我们可能想要在顶端留出一些空白以便和底部和两边对称。将 x 轴的标签放到更远的左侧也是一个更好的想法。我们留下这些缺陷，这样我们就可以时常提及它们——总是会有很多美观细节需要我们继续完善。程序设计艺术的一个重要部分就是知道什么时候停止，将节省出的时间用于更有意义的事情上(比如学习新的技术或者睡觉)。记住：“至善者善之敌。”



15.3 Function 类

Function 图形接口类的定义如下：

```
struct Function : Shape {
    // the function parameters are not stored
    Function(Fct f, double r1, double r2, Point orig,
        int count = 100, double xscale = 25, double yscale = 25);
};
```

Function 是一个 Shape，其构造函数生成很多线段并把它存储在 Shape 中。这些线段是对函数 f 的值的近似。我们在范围 $[r1:r2)$ 内等间隔地计算了 count 次 f 的值：

```
Function::Function(Fct f, double r1, double r2, Point xy,
                  int count, double xscale, double yscale)
// graph f(x) for x in [r1:r2) using count line segments with (0,0) displayed at xy
// x coordinates are scaled by xscale and y coordinates scaled by yscale
{
    if (r2-r1<=0) error("bad graphing range");
    if (count <=0) error("non-positive graphing count");
    double dist = (r2-r1)/count;
    double r = r1;
    for (int i = 0; i<count; ++i) {
        add(Point(xy.x+int(r*xscale),xy.y-int(f(r)*yscale)));
        r += dist;
    }
}
```

xscale 和 yscale 的值分别用于设定 x 坐标和 y 坐标的比例。我们需要设置待显示的数值的比例，使其能适合于窗口的绘制区域。

注意，Function 对象并不存储传递给它的构造函数的值，因此，我们随后无法查询函数的原点，以及使用不同比例重新绘制函数，等等。它实现的功能就是（在它的 Shape 中）存储点并将自身绘制到屏幕上。如果我们需要在构造之后还能改变 Function 这种灵活性，就必须存储那些我们希望修改的值（参见习题 2）。

15.3.1 默认参数

注意，我们赋予 Function 构造函数参数 xscale 和 yscale 初始值的方法。这种初始化值称为默认参数（default argument），如果调用者没有给出参数值，将使用此默认值。例如：

```
Function s(one, r_min, r_max, orig, n_points, x_scale, y_scale);
Function s2(slope, r_min, r_max, orig, n_points, x_scale); // no yscale
Function s3(square, r_min, r_max, orig, n_points); // no xscale, no yscale
Function s4(sqrt, r_min, r_max, orig); // no count, no xscale, no yscale
```

上面的代码等价于：

```
Function s(one, r_min, r_max, orig, n_points, x_scale, y_scale);
Function s2(slope, r_min, r_max, orig, n_points, x_scale, 25);
Function s3(square, r_min, r_max, orig, n_points, 25, 25);
Function s4(sqrt, r_min, r_max, orig, 100, 25, 25);
```

另一种替代方法是提供几个重载函数。我们可以定义 4 个构造函数，而不是定义一个有 3 个默认参数的构造函数：

```
struct Function : Shape { // alternative, not using default arguments
    Function(Fct f, double r1, double r2, Point orig,
            int count, double xscale, double yscale);
    // default scale of y:
    Function(Fct f, double r1, double r2, Point orig,
            int count, double xscale);
    // default scale of x and y:
    Function(Fct f, double r1, double r2, Point orig, int count);
    // default count and default scale of x or y:
    Function(Fct f, double r1, double r2, Point orig);
};
```

定义 4 个构造函数的工作量更多一些，同时对于这个含有 4 个构造函数的版本，默认参数的特性仍然存在，只不过它隐藏在构造函数的定义中，而不是在声明中显式地给出。默认参数经常被用于构

造函数中,但是它对其他类型的函数也适用。注意,只能将末尾的参数定义为默认参数。例如:

```
struct Function : Shape {
    Function(Fct f, double r1, double r2, Point orig,
            int count = 100, double xscale, double yscale); // error
};
```

如果一个参数有一个默认参数值,那么其后的所有参数都必须有一个默认参数值:

```
struct Function : Shape {
    Function(Fct f, double r1, double r2, Point orig,
            int count = 100, double xscale=25, double yscale=25);
};
```

有时候,选择好的默认参数值比较容易。这样的例子包括字符串的默认值(空字符串)和 vector 的默认值(空 vector)。对于其他情况(如 Function)选择一个默认值却不是那么简单;我们通过一些实验和一次失败的尝试之后,找到了现在所使用的这组默认值。记住,你不是必须要提供默认参数,而且如果你发现很难给出一个默认值,简单地将它留给用户来指定即可。

15.3.2 更多的例子

我们增加了一对函数:一个简单的余弦函数(cos)(它来自标准库)和一个用来说明如何组合函数的例子——沿着斜率 $x/2$ 的倾斜余弦函数:

```
double sloping_cos(double x) { return cos(x)+slope(x); }
```

结果如右图所示。对应的代码为:

```
Function s4(cos,r_min,r_max,orig,400,20,20);
s4.set_color(Color::blue);
Function s5(sloping_cos, r_min,r_max,orig,400,20,20);
x.label.move(-160,0);
x.notches.set_color(Color::dark_red);
```

除了增加这两个函数外,我们还移动了 x 轴的标签并稍微改变了它的刻度颜色(只是为了说明如何实现)。

最后,我们用图形显示一个对数函数、一个指数函数、一个正弦函数和一个余弦函数:

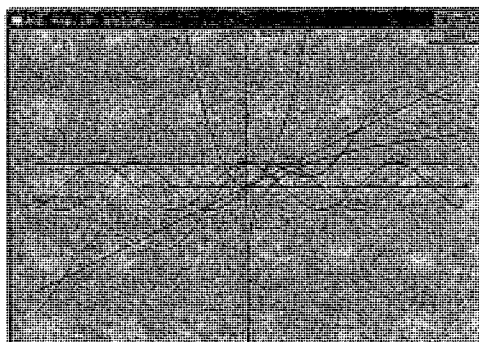
```
Function f1(log,0.000001,r_max,orig,200,30,30); // log() logarithm, base e
Function f2(sin,r_min,r_max,orig,200,30,30); // sin()
f2.set_color(Color::blue);
Function f3(cos,r_min,r_max,orig,200,30,30); // cos()
Function f4(exp,r_min,r_max,orig,200,30,30); // exp() exponential  $e^x$ 
```

因为 $\log(0)$ 是没有定义的(数学上是负无穷大),所以 log 的范围从一个小的正数开始。得到的结果如右图所示。本例中我们用不同颜色区分这些函数而不是为它们添加标签。

$\cos()$ 、 $\sin()$ 和 \sqrt{x} 等标准数学函数都是在标准库头文件 `<cmath>` 中声明的。标准数学函数的列表请参见 24.8 节和附录 B.9.2。

15.4 Axis 类

当我们显示数据时,就需要使用 Axis(例如 15.6.4 节),因为一个没有比例信息的图形常常令人怀疑。一个 Axis 由一条线、在这条线上的一系列“刻度”和一个文本标签组成。Axis 的构造



函数计算坐标线和(可选的)这条线上作为刻度的一些线:

```
struct Axis : Shape {
    enum Orientation { x, y, z };
    Axis(Orientation d, Point xy, int length,
        int number_of_notches=0, string label = "");

    void draw_lines() const;
    void move(int dx, int dy);
    void set_color(Color c);

    Text label;
    Lines notches;
};
```

其中 label 和 notches 对象定义为公有的,便于用户处理它们。例如,你可以为刻度设置一个不同的颜色或者使用 move() 将 label 对象移动到更合适的位置上。Axis 给出了一个由若干半独立对象组成的对象的例子。

Axis 的构造函数负责放置坐标线,并且如果 number_of_notches 大于 0 的话,它还负责添加“刻度”。

```
Axis::Axis(Orientation d, Point xy, int length, int n, string lab)
    :label(Point(0,0),lab)
{
    if (length<0) error("bad axis length");
    switch (d){
    case Axis::x:
    {
        Shape::add(xy);           // axis line
        Shape::add(Point(xy.x+length,xy.y));

        if (1<n) {                // add notches
            int dist = length/n;
            int x = xy.x+dist;
            for (int i = 0; i<n; ++i) {
                notches.add(Point(x,xy.y),Point(x,xy.y-5));
                x += dist;
            }
        }

        label.move(length/3,xy.y+20); // put the label under the line
        break;
    }
    case Axis::y:
    {
        Shape::add(xy);           // a y axis goes up
        Shape::add(Point(xy.x,xy.y-length));

        if (1<n) {                // add notches
            int dist = length/n;
            int y = xy.y-dist;
            for (int i = 0; i<n; ++i) {
                notches.add(Point(xy.x,y),Point(xy.x+5,y));
                y -= dist;
            }
        }

        label.move(xy.x-10,xy.y-length-10); // put the label at top
        break;
    }
}
```

```

    }
    case Axis::z:
        error("z axis not implemented");
    }
}

```

与很多实际的代码相比,这个构造函数非常简单,但是请仔细阅读,因为它并不是十分简单,并且还阐述了一些有用的技术。注意,我们如何将线存储在 Axis 的 Shape 部分(使用 Shape::add()),而将刻度存储在一个独立的对象(notches)中。通过这种方式,我们可以独立地处理线和刻度;例如,我们可以将它们设置为不同的颜色。类似地,标签放置在相对于坐标轴的固定位置上,但因为它也是一个独立的对象,我们总是可以将它移动到一个更好的位置。我们使用枚举类型 Orientation 来为用户提供一个方便的并且不易出错的符号。

因为 Axis 有三个部分,所以当我们希望把 Axis 作为一个整体来操作的时候,必须提供相应的函数。例如:

```

void Axis::draw_lines() const
{
    Shape::draw_lines();
    notches.draw(); // the notches may have a different color from the line
    label.draw();   // the label may have a different color from the line
}

```

我们使用 draw() 而不是 draw_lines() 函数来绘制 notches 和 label,这样我们可以使用它们各自的颜色。线被存储在 Axis::Shape 中,并且使用存储在相同位置的颜色。

我们可以分别为线、刻度和标签设置颜色,但是从风格上来说,一般最好不要这样做,因此我们提供了一个函数将这三部分设置为相同的颜色:

```

void Axis::set_color(Color c)
{
    Shape::set_color(c);
    notches.set_color(c);
    label.set_color(c);
}

```

类似地,Axis::move() 同时移动 Axis 的所有部分:

```

void Axis::move(int dx, int dy)
{
    Shape::move(dx,dy);
    notches.move(dx,dy);
    label.move(dx,dy);
}

```

15.5 近似

本节中我们给出图形化函数的另一个例子:我们“动态地展示”一个指数函数的计算过程。其目的是帮助你获得数学函数的感性认识(如果你还没有),说明使用图形来显示计算的方式,给出一些供你阅读的代码,最后对计算中的常见问题给出警告。

计算一个指数函数的一种方法是来计算如下序列:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

这个序列的项越多,得到的 e^x 的值就越精确;也就是说,我们计算的项越多,得到的结果就有更多的正确位数。我们要做的就是计算这个序列,同时每计算一项就图形化显示其结果。这里的感叹号代表其通常的数学意义:阶乘;也就是说,我们要按顺序图形化显示如下函数:

```

exp0(x) = 0      // no terms
exp1(x) = 1      // one term
exp2(x) = 1+x    // two terms; pow(x,1)/fac(1)==x
exp3(x) = 1+x+pow(x,2)/fac(2)
exp4(x) = 1+x+pow(x,2)/fac(2)+pow(x,3)/fac(3)
exp5(x) = 1+x+pow(x,2)/fac(2)+pow(x,3)/fac(3)+pow(x,4)/fac(4)
...

```

每个函数都比前一个更接近于 e^x 。此处， $\text{pow}(x, n)$ 是返回 x^n 值的标准库函数。标准库中没有阶乘函数，所以我们必须自己定义：

```

int fac(int n)    // factorial(n); n!
{
    int r = 1;
    while (n>1) {
        r*=n;
        --n;
    }
    return r;
}

```

$\text{fac}()$ 的另一种实现方法，参见习题 1。给定 $\text{fac}()$ ，我们就可以按如下方式计算出第 n 项：

```
double term(double x, int n) { return pow(x,n)/fac(n); } // nth term of series
```

给定 $\text{term}()$ ，计算指数函数的 n 项序列精度就很简单了：

```

double expe(double x, int n)    // sum of n terms for x
{
    double sum = 0;
    for (int i=0; i<n; ++i) sum+=term(x,i);
    return sum;
}

```

我们如何用图形显示它呢？从程序设计的角度看，难点在于我们的图形类 `Function`，它使用的是带有一个参数的函数，而 $\text{expe}()$ 有两个参数。在 C++ 中，到目前为止还没有此问题的完美解决方法。因此对目前的情况，我们使用了一个简单的并不完美的解决方法（参见习题 3）。我们可以将精度 n 从参数列表中拿出来，定义一个变量来表示它：

```

int expN_number_of_terms = 10;

double expN(double x)
{
    return expe(x,expN_number_of_terms);
}

```

现在利用 $\text{expN}(x)$ 可以计算由 $\text{expN_number_of_terms}$ 的值指定精度的指数函数了。让我们使用它来生成一些图形。首先，我们提供一些坐标轴和“实际”指数函数——标准库函数 $\text{exp}()$ ，这样就可以看到我们使用 $\text{expN}()$ 得到的近似值与真实值的接近程度：

```

Function real_exp(exp,r_min,r_max,orig,200,x_scale,y_scale);
real_exp.set_color(Color::blue);

```

然后，可以通过每次增加近似值的项数 n 来循环处理一系列近似值：

```

for (int n = 0; n<50; ++n) {
    ostringstream ss;
    ss << "exp approximation; n==" << n ;
    win.set_label(ss.str().c_str());
    expN_number_of_terms = n;
    // get next approximation:
    Function e(expN,r_min,r_max,orig,200,x_scale,y_scale);
    win.attach(e);
}

```

```

win.wait_for_button();
win.detach(e);
}

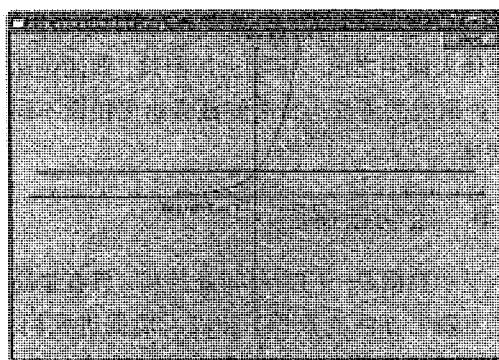
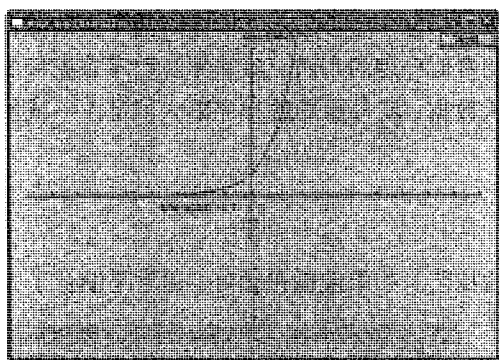
```

注意，这个循环最后的 `detach(e)`。Function 对象 `e` 的作用域是在 `for` 循环体内。因此每执行一次循环我们就得到一个名字为 `e` 的新 Function 对象，而每个循环结束这个 `e` 就消亡了，下一次会被新的 `e` 代替。因为 `e` 将会销毁，所以窗口必须丢弃那个旧的 `e`。`detach(e)` 就是保证窗口不会试图绘制一个已经被销毁的对象。

此代码首先显示一个窗口，其中只显示了坐标轴和蓝色的“实际”指数函数：

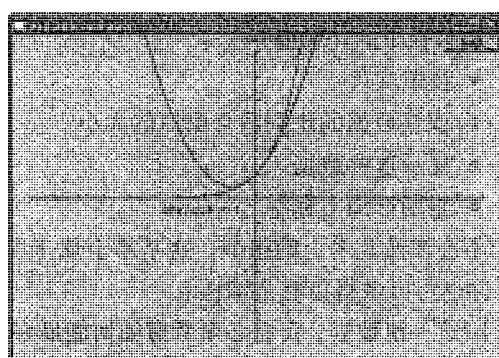
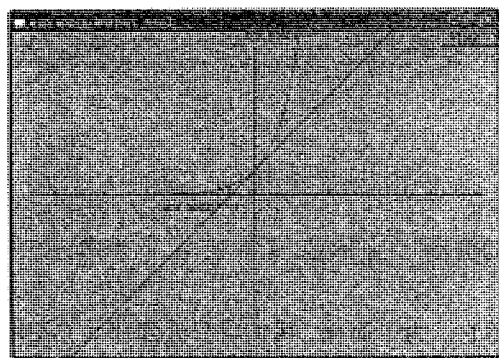
我们知道 $\exp(0)$ 的值为 1，所以这条蓝色的“实际指数”与 y 轴相交于点 $(0, 1)$ 。

如果你仔细观察，你会发现我们实际用一条黑色的线在 x 轴正上方绘制了零个项的近似值 ($\exp(0) = 1$)。点击“Next”，我们得到只使用一项的近似值。注意我们将项数显示在窗口标题栏中：



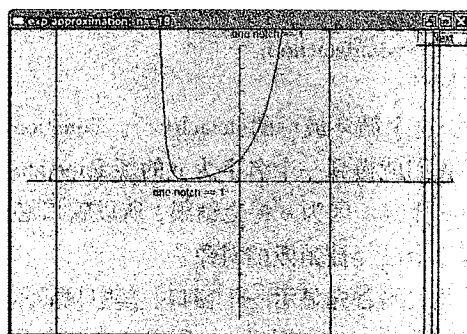
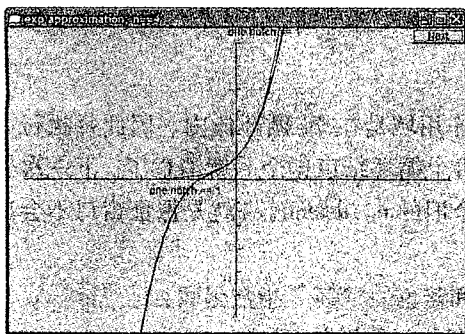
这里的函数是 $\exp(1) = 1$ ，该近似值只使用了序列中的一项。它准确地和指数函数相交于 $(0, 1)$ ，但我们还可以做得更好：

使用两项 $(1 + x)$ ，我们得到和 y 轴相交于点 $(0, 1)$ 的对角线。使用 3 项 $(1 + x + \text{pow}(x, 2)/\text{fac}(2))$ ，我们可以看到两条曲线开始汇聚：



使用 10 项可以得到更好的结果，特别是对于大于 -3 的值：

如果你不仔细思考这个问题的话，你可能会认为可以简单地通过使用越来越多的项来得到越来越好的近似值。然而，这是有极限的，当超过 13 项之后会发生一些奇怪的事情。首先，近似值开始慢慢变差，而在 18 项时会出现一些竖直线：



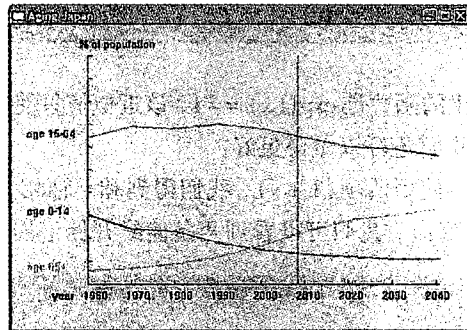
记住，浮点算术并不是纯粹的数学运算。用浮点数表示实数，只能得到和固定位数一样的近似结果。问题在于我们的运算开始产生一些不能表示为 `double` 类型的值，导致结果开始偏离数学上正确的结果。更多信息请参见第 24 章。

最后这幅图片是“看起来正确”不等同于“通过测试”这一原则的一个很好的例子。在把程序交给他人使用之前一定要进行测试，即便是那些起初看似合理的东西。除非你对程序有更深入的理解，否则稍微延长运行时间或者给出稍微不同的输入数据，就可能导致程序陷入混乱——就像本例这样。

15.6 绘制数据图

显示数据是一门需要很高技巧，具有很高价值的技艺。如果能做得很好，通常是结合了技术和艺术两个方面的知识，能极大地促进我们对复杂现象的理解。但是，它也使图形化变成一个巨大的领域，而且其大部分和程序设计技术无关。这里，我们仅仅展示一个简单的例子，它显示从一个文件中读取的数据。这些数据给出了近一个世纪以来日本人的年龄构成。2008 年以后的数据是预测的结果如右图所示。我们将使用这个例子来讨论显示这种数据涉及的程序设计问题：

- 读取文件
- 调整数据的比例适合窗口的大小
- 显示这些数据
- 给图形加上标签



我们不会涉及艺术上的细节。这基本上属于“简单的图形”，而不是“图形艺术”。当然，如果需要，你也可以做得更有艺术性。

给定一组数据，我们必须考虑如何能最好地显示它。为了简化，我们将只处理那些方便用二维显示的数据，不过这也正是大部分人需要处理的数据中最大的一部分。注意，那些柱状图、饼图和类似的流行显示方式，其实也都是用一种有趣的二维方式显示的。三维数据的处理通常也是生成一系列二维图像，或者在一个窗口中叠加几张二维图形（如“日本人的年龄”的例子），或者对单个点添加信息标签等。如果要超出这些方法，我们就必需编写新的图形类或者采用其他的图形库。

所以，我们的数据是基本的数值对，例如 `(year, number of children)`。如果有更多的数据，例

如(year, number of children, number of adults, number of elderly), 我们只是需要确定哪一对或者哪几对数据是需要绘制的。在我们的例子中, 我们只是图形显示(year, number of children)、(year, number of adults)和(year, number of elderly)。

我们有很多方式看待一组(x, y)数值对。当考虑如何用图形显示这样一组数据的时候, 重要的是要考虑一个数值是否在某种意义上是另一个数值的函数。例如, 对于一个(year, steel production)对, 很明显可以把钢产量作为年份的一个函数并以一条连续的线显示数据。可以用 Open_polyline(参见 13.6 节)显示这类数据。如果 y 不应该被看做 x 的函数, 例如(gross domestic product per person, population of country), 可以用 Marks(参见 13.15 节)绘制相互独立的点。

现在, 回到日本人年龄分布的例子。

15.6.1 读取文件

年龄分布文件由很多行组成, 例如:

```
(1960 : 30 64 6 )
(1970 : 24 69 7 )
(1980 : 23 68 9 )
```

冒号后面的第一个数字是儿童(0 ~ 14 岁)在总人数中的百分比, 第二个是成年人(15 ~ 64 岁)的百分比, 第三个是老年人(65 岁以上)的百分比。我们的目标就是读出这些数据。注意, 数据的格式有点不规则。与往常一样, 我们必须处理这些细节。

为了简化这个任务, 我们首先定义一个保存数据项的 Distribution 类型和一个读取这些数据项的输入操作符。

```
struct Distribution {
    int year, young, middle, old;
};

istream& operator>>(istream& is, Distribution& d)
// assume format: ( year : young middle old )
{
    char ch1 = 0;
    char ch2 = 0;
    char ch3 = 0;
    Distribution dd;

    if (is >> ch1 >> dd.year
        >> ch2 >> dd.young >> dd.middle >> dd.old
        >> ch3) {
        if (ch1 != '(' || ch2 != ':' || ch3 != ')') {
            is.clear(ios_base::failbit);
            return is;
        }
    }
    else
        return is;
    d = dd;
    return is;
}
```

这是第 10 章中的概念的一个直接应用。如果你不熟悉这段代码, 请复习第 10 章。我们不是必须要定义一个 Distribution 类型和一个 >> 操作符。然而, 与“只是读取数字并用图形显示它们”的蛮力方法相比, 这样做可以简化代码。我们使用 Distribution 将代码划分为有助于理解和调试的几个逻辑部分。不要觉得引入新类型“只是为了使代码清晰”。类的定义和使用能使代码更加直接地

对应我们对概念的思考。即使对那些仅仅在代码局部区域中使用的“小”概念也应这么做，例如一行数据表示一年的年龄分布，这会很有帮助。

给定 Distribution，读取循环可这样设计：

```
string file_name = "japanese-age-data.txt";
ifstream ifs(file_name.c_str());
if (!ifs) error("can't open ", file_name);

// ...

Distribution d;
while (ifs >> d) {
    if (d.year < base_year || end_year < d.year)
        error("year out of range");
    if (d.young + d.middle + d.old != 100)
        error("percentages don't add up");
    // ...
}
```

也就是说，我们试图打开文件“japanese - age - data. txt”，如果找不到这个文件就退出程序。不将文件名“硬编码”到源代码中会更好一些，但我们考虑到这只是一个“一次性”的小程序，所以没有采用这种更好的方法，它适合长期使用的应用程序，但会增加这个小程序的负担。另一方面，我们确实是把 japanese-age-data. txt 存在一个命名的 string 变量中，如果我们将此程序或它的一部分用做其他用途，程序也很容易修改。

读取循环检查所读年份是否在预期的范围内，同时检查百分比之和是否为 100。这是一个基本的数据检查，因为 >> 检查了每个单独的数据项的格式，我们不用在主循环中再做更多的检查。

15.6.2 一般布局

那么我们想在屏幕上显示什么呢？你可以从 15.6 节的开始看到答案。这些数据看起来需要 3 个 Open_polyline，分别对应 3 个年龄组。因为这些图形需要添加标签，所以我们决定为每条线在窗口的左侧写一个“标题”。在这种情况下，这种方式看起来比将标签放在线上某个位置的方式更为清晰。另外，我们使用颜色来区分图形并与标签关联。

我们想用年份来标注 x 轴。2008 年处的那条竖直线表明后面的图形是根据预测数据绘制的。

我们决定使用窗口标签作为我们图形的标题。

让图形化的代码既正确又美观是非常棘手的。一个主要原因是我们需要做很多有关尺寸和偏移量的高精度计算。为了简化，我们开始先定义一组符号常量来表示对屏幕空间的使用方式：

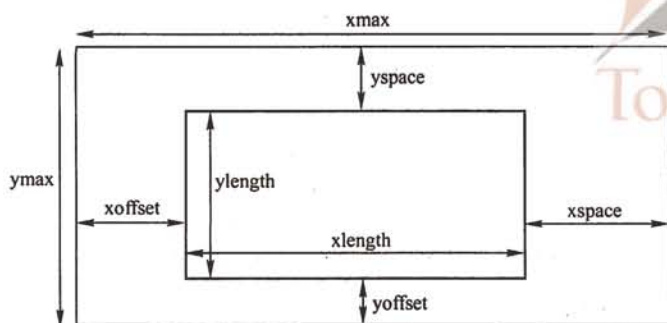
```
const int xmax = 600; // window size
const int ymax = 400;

const int xoffset = 100; // distance from left-hand side of window to y axis
const int yoffset = 60; // distance from bottom of window to x axis

const int xspace = 40; // space beyond axis
const int yspace = 40;

const int xlength = xmax - xoffset - xspace; // length of axes
const int ylength = ymax - yoffset - yspace;
```

基本上，这里定义了一个矩形空间（窗口）及其内部的另一个矩形（通过坐标轴定义），如下图所示：



如果没有这样一个表明窗口中的事物位置的“示意图”和用于定义位置的符号常量，当程序输出不能反映所要求的结果时，我们将会迷失方向并且感到无助。

15.6.3 数据比例

接下来，需要定义怎样才能让数据适合这个空间。我们通过按比例缩放数据来达到这个目的，使数据适合由坐标轴定义的空间。因此，我们需要使用比例因子，即数据范围和坐标轴范围之间的比值。

```
const int base_year = 1960;
const int end_year = 2040;

const double xscale = double(xlength)/(end_year-base_year);
const double yscale = double(ylength)/100;
```

比例因子 (xscale 和 yscale) 应该是浮点数——否则我们的运算将会面对严重的舍入误差。为了避免整数除法，在做除法之前先把长度数据转换成 double 类型 (参见 4.3.3 节)。

现在，我们可以通过减去基准值 (1960)，使用 xscale 进行比例缩放，并加上 xoffset，将一个数据点放到 x 轴上。按同样的方式可以处理 y 轴上的数据。不过，当试图重复做这件事情的时候，我们发现很难记得非常清楚。这可能是一个不太重要的运算，但它是需要技巧和时间的。为了简化代码并减小出错的机会 (尽量减少令人沮丧的调试工作)，我们定义了一个很小的类来完成这个运算。

```
class Scale { // data value to coordinate conversion
    int cbase; // coordinate base
    int vbase; // base of values
    double scale;
public:
    Scale(int b, int vb, double s) : cbase(b), vbase(vb), scale(s) {}
    int operator()(int v) const { return cbase + (v-vbase)*scale; }
};
```

我们需要一个类，因为这个运算依赖 3 个常量值，而我们又不愿意总是不必要地重复它们。给定这个类，我们可定义：

```
Scale xs(xoffset, base_year, xscale);
Scale ys(ymax-yoffset, 0, -yscale);
```

注意，我们是如何使 ys 的比例因子变为负值，以反映 y 坐标向下增长的——我们通常用图形上更高的点表示更大的数值。现在，我们可以使用 xs 将一个年份转换为 x 坐标。类似地，可以使用 ys 将一个百分数转换为 y 坐标。

15.6.4 构造数据图

最后，我们具备了采用合理方式来编写图形化代码的所有先决条件。首先我们创建窗口并放置坐标轴：

```

Window win(Point(100,100),xmax,ymax,"Aging Japan");

Axis x(Axis::x, Point(xoffset,ymax-yoffset), xlength,
        (end_year-base_year)/10,
        "year 1960 1970 1980 1990 "
        "2000 2010 2020 2030 2040");
x.label.move(-100,0);

Axis y(Axis::y, Point(xoffset,ymax-yoffset), ylength, 10,"% of population");

Line current_year(Point(xs(2008),ys(0)),Point(xs(2008),ys(100)));
current_year.set_style(Line_style::dash);

```

坐标轴的交点 `Point(xoffset, ymax-yoffset)` 表示 (1960, 0)。注意, 这里是如何放置刻度来反映数据的。在 y 轴上有 10 个刻度, 每一个刻度代表 10% 的人口。在 x 轴上, 每个刻度代表 10 年, 而具体的刻度数值是通过 `base_year` 和 `end_year` 计算得来的, 因此, 如果我们改变这个范围, 该坐标轴也会自动地重新计算。这是在代码中避免使用“魔数”的一个优点。但是, 在 x 轴上的标签违反了规则: 它只是用手动调整标签字符串的方式得到的结果, 直到数字正好在对应的刻度下面。更好的方法是针对一组单独的“刻度”给出一组对应的标签。

请注意标签字符串的格式, 我们使用了两个相邻的文字常量字符串:

```

"year 1960 1970 1980 1990 "
"2000 2010 2020 2030 2040"

```

相邻的文字常量字符串会被编译器连接起来, 相当于

```
"year 1960 1970 1980 1990 2000 2010 2020 2030 2040"
```

这是一个用来布局长字符串从而使代码更可读的有用“技巧”。

`current_year` 对象是一条分割已知数据和预测数据的垂直线。注意, 如何使用 `xs` 和 `ys` 来正确地布局 and 按比例缩放这条线。

给定坐标轴, 我们就可以处理数据了。定义 3 个 `Open_polyline` 对象, 在读取循环中向它们添加数据:

```

Open_polyline children;
Open_polyline adults;
Open_polyline aged;

Distribution d;
while (ifs>>d) {
    if (d.year<base_year || end_year<d.year) error("year out of range");
    if (d.young+d.middle+d.old != 100)
        error("percentages don't add up");
    int x = xs(d.year);
    children.add(Point(x,ys(d.young)));
    adults.add(Point(x,ys(d.middle)));
    aged.add(Point(x,ys(d.old)));
}

```

`xs` 和 `ys` 的使用可以让数据的放置和按比例缩放变得容易。像 `Scale` 这种“很小的类”对于简化符号和避免不必要的重复是非常重要的, 能够提高程序的可读性和正确性。

为了使图形更具有可读性, 我们为每一个图形添加标签并设置颜色:

```

Text children_label(Point(20,children.point(0).y),"age 0-14");
children.set_color(Color::red);
children_label.set_color(Color::red);

Text adults_label(Point(20,adults.point(0).y),"age 15-64");

```

```
adults.set_color(Color::blue);
adults_label.set_color(Color::blue);

Text aged_label(Point(20,aged.point(0).y),"age 65+");
aged.set_color(Color::dark_green);
aged_label.set_color(Color::dark_green);
```

最后，我们需要把不同的 Shape 对象添加到 Window 对象中，然后启动 GUI 系统(参见 14.2.3 节)：

```
win.attach(children);
win.attach(adults);
win.attach(aged);

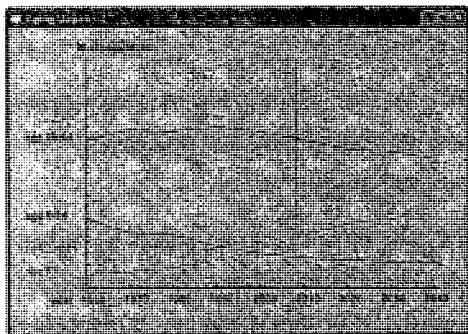
win.attach(children_label);
win.attach(adults_label);
win.attach(aged_label);

win.attach(x);
win.attach(y);
win.attach(current_year);

gui_main();
```

所有代码都可以放在 main() 中，但我们认为将辅助类 Scale 和 Distribution 与 Distribution 的输入操作符一起放在 main() 之外更好。

假如你已经忘记我们正在生成什么图形，我们再次给出输出结果，如右图所示。



简单练习

函数图形化显示练习：

1. 创建一个标签为“Function graphs”的空的大小为 600 × 600 的 Window。
2. 注意，你可能需要参照“installation of FLTK”(可从课程网站下载)中的说明创建一个项目，并设置项目属性。
3. 将 Graph.cpp 和 Window.cpp 移入你的项目中。
4. 向窗口中加入一条 x 坐标轴和一条 y 坐标轴，长度均为 400，标签为“1 == 20 pixels”，每隔 20 个像素画一个刻度。两条坐标轴相交于(300, 300)。
5. 将两条坐标轴都设置为红色。

在下面的练习中，对每个图形化显示的函数都使用一个独立的 Shape：

1. 图形化显示函数 `double one(double x) { return 1; }`，参数范围 $[-10, 11]$ ，原点(0, 0)位于坐标点(300, 300)，显示 400 个点(400 个函数值)，(在窗口中)不缩放。
2. 将 x 轴和 y 轴缩放比例均改为 20。
3. 之后所有练习均使用当前的参数范围和缩放比例等设置。
4. 将 `double slope(double x) { return x/2; }` 的图形显示加到窗口中。
5. 用 Text 对象“x/2”为斜线添加标签，添加位置为斜线的左下角。
6. 将 `double square(double x) { return x * x; }` 的图形显示加入窗口中。
7. 向窗口中加入余弦曲线(不要编写一个新函数)。
8. 将余弦曲线设置为蓝色。
9. 编写函数 `sloping_cos()`，将余弦曲线添加到 `slope()` (如上定义)上，并将此函数的图形显示加入窗口。

类定义练习：

1. 定义一个 struct Person，包含一个类型为 string 的名字(name)和类型为 int 的年龄(age)。

2. 定义一个类型为 `Person` 的变量, 用“Goofy”和 63 对其初始化, 并将其输出到屏幕(cout)。
3. 为 `Person` 定义输入操纵符(>>)和输出操纵符(<<), 从键盘(cin)读入一个 `Person` 值, 并将其写到屏幕(cout)。
4. 为 `Person` 定义一个构造函数, 初始化 `name` 和 `age`。
5. 将 `Person` 的描述改为 `private`, 并提供 `const` 成员函数 `name()` 和 `age()` 实现名字和年龄的读取。
6. 修改 >> 和 <<, 使之适应重新定义过的 `Person`。
7. 修改构造函数, 检查 `age`, 保证它在范围[0:150]之内, 检查 `name`, 保证它不包含: ; " ' [] * & ^ % \$ # @ !。如果发生错误, 使用 `error()`。测试这个构造函数。
8. 从标准输入(cin)读取一组 `Person`, 存入一个 `vector <Person>` 中; 将它们写到屏幕(cout)。用正确和错误的输入数据测试这个程序。
9. 修改 `Person` 的描述, 用 `first_name` 和 `second_name` 代替 `name`。如果用户未提供 `first_name` 或 `second_name`, 则给出一个错误。相应修改 >> 和 <<。测试新的类。

思考题

1. 接受一个参数的函数是怎样的?
2. 什么情况下你会使用(连续的)线表示数据? 什么情况下你会使用点?
3. 什么函数(数学公式)定义一条斜线?
4. 什么是抛物线?
5. 如何生成 x 轴和 y 轴?
6. 什么是默认参数? 什么时候使用默认参数?
7. 如何把函数叠加在一起?
8. 如何给一个图形化的函数加上颜色和标签?
9. 我们说一个序列近似一个函数, 这是什么意思?
10. 为什么在编写代码绘制图形之前需要画出它的布局草图?
11. 如何按比例缩放图形使得输入恰好适合显示区域?
12. 如何按比例缩放输入, 可以避免试验和误差?
13. 为什么要格式化输入, 而不只是让文件包含那些“数字”?
14. 如何设计图形的总体布局? 如何在代码中反映出来?

术语

近似 函数 屏幕布局 默认参数 缩放比例

习题

1. 下面是定义阶乘函数的另一种方式:

```
int fac(int n) { return n>1 ? n*fac(n-1) : 1; } // factorial n!
```

 对于 `fac(4)`, 因为 $4 > 1$, 所以第一次调用会执行 $4 * \text{fac}(3)$, 接下来是 $4 * 3 * \text{fac}(2)$, 然后是 $4 * 3 * 2 * \text{fac}(1)$, 即 $4 * 3 * 2 * 1$ 。试着体会它是怎么执行的。一个函数调用它自身称为递归(recursive)。在 15.5 节中给出的另一种实现方式称为迭代(iterative), 因为它对一系列数值反复进行计算(使用 `while`)。验证递归函数 `fac()` 的执行过程, 并通过计算 0, 1, 2, 3, 4, ..., 20 的阶乘, 验证是否与迭代函数 `fac()` 有相同的执行结果。你更倾向于 `fac()` 的哪种实现? 为什么?
2. 定义一个 `Fct` 类, 除了存储构造函数的参数之外, 它与 `Function` 类一样。给 `Fct` 类提供“复位”(reset)操作, 从而可以重复利用它对不同的范围、不同的函数等生成图形。
3. 修改上面的 `Fct` 类, 使其带有一个额外的参数来控制精度或其他内容。为了更加灵活, 可将该参数的类型设置为模板参数。
4. 在一个图上绘制正弦(`sin()`)、余弦(`cos()`)、正弦与余弦的和($\sin(x) + \cos(x)$)以及正弦平方与余弦平方的和($\sin(x) * \sin(x) + \cos(x) * \cos(x)$)。注意要提供坐标轴和标签。
5. “动态显示”(像 15.5 节中那样)序列 $1 - 1/3 + 1/5 - 1/7 + 1/9 - 1/11 + \dots$ 。它是著名的莱布尼茨序列, 收

敛到 $\pi/4$ 。

6. 设计并实现一个柱状图类。它的基本数据是一个保存 N 个数值的 `vector < double >`，每个数值用一个矩形形状的“柱”(bar)表示，其高度代表对应的数值。
7. 细化该柱状图类，实现对图本身和每一个单独的柱添加标签的功能，并允许使用颜色。
8. 有一个以厘米为单位的身高(取整到最接近的 5 cm 的整数倍)和身高为对应值的人数构成的集合：(170, 7), (175, 9), (180, 23), (185, 17), (190, 6), (195, 1)。如何图形化这些数据？如果你想不到更好的方法，做一个柱状图。记得提供坐标轴和标签。把数据放在一个文件中，并从该文件读取这些数据。
9. 找另一个身高的数据集合，然后用前面练习中的程序图形化这些数据。例如，从网上搜索“身高分布”或者“美国人的身高”，并忽略没用的内容，或者向你的朋友询问他们的身高。理想情况下，你不需要对新的数据集做任何改变。关键思路是计算出数据的缩放比例。从输入中读取标签也有助于代码重用尽量少地修改代码。
10. 什么样的数据不适合用线图或者柱状图表示？寻找一个例子，给出显示它的一种方法(例如，一个标记点的集合)。
11. 计算两个或者更多地点(例如，英格兰的剑桥和马萨诸塞州的剑桥；有很多城镇叫做“剑桥”)一年中每个月的平均最高气温，并将它们显示在一张图上。注意坐标轴、标签、颜色的使用等。



附言

数据的图形表示是重要的。与一组数据相比，我们更容易理解由数据制作而得到的图。当需要绘制图形的时候，大部分人都会使用其他人的代码——函数库。这些库是如何构造的呢？而如果你手头没有这样一个库，又该如何做呢？“一个普通的绘图工具”之下的基本思想是什么呢？现在你已经知道：它不是神秘的魔术或者脑外科手术。我们只讨论了二维图形；而三维图形化表示在科学、工程、市场等领域同样非常有用，甚至更加有趣。将来如有恰当的时机，请仔细研究它们！

第 16 章 图形用户界面

“计算不再是指计算机，而是生活。”

——Nicholas Negroponte

图形用户界面(GUI)允许用户通过点击按钮、选择菜单、以不同的方式输入数据以及在屏幕上显示文本和图形等方式与程序进行交互。这正是我们与电脑以及网站交互时经常采用的方式。在本章中，我们将介绍编写代码来定义和控制 GUI 应用的基本方法。特别地，我们将介绍如何编写代码，通过回调函数与屏幕上的实体进行交互。我们的 GUI 工具是建立在系统工具之上的。附录 E 给出了更低层次的特性和接口，这些特性和接口使用了第 17 章和第 18 章中介绍的特性和技术。在本章中，我们将注意力集中在使用方法上。

16.1 用户界面的选择

每个程序都有用户接口。在小装置上运行的程序的接口可能局限于从几个按钮输入和用一个闪光信号灯输出。而其他的计算机仅仅通过一条线就可以连接到外面的世界。这里，我们将考虑一般的情况，即程序是和一个看着屏幕、使用键盘和定点设备(如鼠标)的用户进行交互。在这种情况下，程序员有三种主要的选择：

- 使用控制台输入/输出：对于专业技术工作而言，这是一种强有力的方式，输入是简单的、文本式的，由一些命令和短数据项(比如文件名或者简单的数值)组成。如果输出也是文本式的，我们可以将它显示在屏幕上或者存储在文件中。C++ 的标准库 `iostream`(参见第 10~11 章)为这种方式提供了适合、方便的机制。如果需要图形输出，我们可以使用一个图形显示库(参见第 12~15 章)，不需要对我们的程序设计风格进行明显的修改。
- 使用图形用户界面(GUI)库：用户的交互基于操纵屏幕对象的方式(定点、点击、拖放、悬停等)。通常(但不总是)，这种方式总是伴随着信息的高度图形化显示。任何用过现代计算机的人都能举出一些体现这种方式便利性的例子。任何希望感受 Windows/Mac 应用的人都需要使用 GUI 交互方式。
- 使用网络浏览器界面：对于这种方式，我们需要使用一种标记(布局)语言，例如 HTML，通常还会使用一种脚本语言。阐述如何实现这种方式已经超出了本书的讨论范围，但一般来说，它是有远程访问需求的应用程序的理想模式。在这种方式下，程序和屏幕之间的通信还是文本方式的(使用字符流)。浏览器是一个 GUI 应用程序，它负责将其中一些文本信息转换成图形元素，并将鼠标点击等转换成文本数据传递回程序。

对于很多人来说，GUI 的使用就是现代程序设计的本质，而且有时与屏幕对象的交互被认为是程序设计的核心概念。我们并不同意这种观点：GUI 是一种 I/O 形式，应用程序的主要逻辑和 I/O 相互分离是软件设计的主要观点之一。无论是否可能，我们更愿意在主要程序逻辑和用来进行输入/输出的部分之间建立起一个清晰的接口。这种分离机制允许我们通过改变程序呈现给用户的方式，来将程序移植到不同的 I/O 系统中，更重要的是，这种机制允许我们将程序逻辑和用户交互分离开来考虑。

也就是说,从多个角度来看,GUI 都是非常重要和有趣的。本章既研究如何把图形元素集成到我们的应用程序中,同时也探索如何防止对界面的过度关注而影响我们的思维。

16.2 “Next”按钮

如何提供一个像第 12 ~ 15 章的例子中用于驱动图形显示的“Next”按钮呢?在那里,我们使用窗口中的一个按钮绘制图形。很明显,这是一种简单的 GUI 程序设计模式。实际上,因为它过于简单,以至于有些人可能会争论它是不是一个“真正的 GUI”。无论如何,让我们看看它是怎样实现的,因为它将引领我们直接进入所有人都认可的真正的 GUI 程序设计。

第 12 ~ 15 章中代码的典型结构如下:

```
// create objects and/or manipulate objects, display them in Window win:
win.wait_for_button();

// create objects and/or manipulate objects, display them in Window win:
win.wait_for_button();

// create objects and/or manipulate objects, display them in Window win:
win.wait_for_button();
```

每当运行到 `wait_for_button()`,就可以在屏幕上看到要显示的对象,直到我们点击按钮来得到程序下一部分的输出。从程序逻辑的观点来看,这种方式与逐行输出到屏幕(控制台窗口),在某处停下来,然后再从键盘接收输入的程序没有区别。例如:

```
// define variables and/or compute values, produce output
cin >> var; // wait for input

// define variables and/or compute values, produce output
cin >> var; // wait for input

// define variables and/or compute values, produce output
cin >> var; // wait for input
```

但是从实现的观点来看,这两种程序截然不同。当你的程序执行到 `cin >> var` 时,它停下来并且等待“系统”读取你输入的字符。然而,监视屏幕并且跟踪鼠标的系统(图形用户界面系统)运行在一种截然不同的模式下:GUI 跟踪鼠标的位置和用户对鼠标所做的操作(点击等)。当你的程序需要一个动作时,它必须

- 告诉 GUI 关心哪些事情(例如,“有人点击了‘Next’按钮”)。
- 告诉 GUI 当有人做这些事的时候,应该如何处理。
- 在 GUI 检测到程序感兴趣的动作之前一直处于等待状态。

与控制台程序的不同之处在于,GUI 并不是简单地返回我们的程序;它的设计目标是对很多不同的用户动作给出不同的响应,例如点击很多按钮中的某一个、改变窗口的尺寸、当窗口被其他内容挡住之后重新绘制窗口以及弹出“弹出式”菜单等。

首先,我们只是想说,“当有人按下我的按钮时请将我唤醒”;也就是说,“当有人点击鼠标按钮,且光标位置在我的按钮图形显示的矩形区域内时,请继续执行我的程序”。这是我们能够想象到的最简单的动作。然而,这样的操作并不是由“系统”提供的,所以我们必须自己编写代码。观察它的实现方法是理解 GUI 程序设计的第一步。

16.3 一个简单的窗口

实际上,“系统”(GUI 库和操作系统的组合)不断跟踪鼠标的位置及其按钮是否被按下。一

个程序可以关注屏幕的某个区域，并请求“系统”在某些被关注的事件发生时调用某个函数。在本例中，我们请求系统当用户在“我们的按钮”上点击鼠标时，调用我们的一个函数（“回调函数”）。要完成这些，必须

- 定义一个按钮
- 显示按钮
- 定义一个 GUI 可以调用的函数
- 将定义的按钮和函数告知 GUI
- 等待 GUI 调用我们的函数

下面进行具体实现。按钮是 Window 的一部分，所以我们（在 Simple_window.h 中）定义了类 Simple_window，这个类包括数据成员 next_button：

```
struct Simple_window : Graph_lib::Window {
    Simple_window(Point xy, int w, int h, const string& title);

    void wait_for_button();    // simple event loop
private:
    Button next_button;        // the "Next" button
    bool button_pushed;        // implementation detail

    static void cb_next(Address, Address); // callback for next_button
    void next(); // action to be done when next_button is pressed
};
```

很显然，类 Simple_window 派生自 Graph_lib 库的 Window 类。所有的窗口都必须直接或者间接地派生自 Graph_lib::Window 类，因为它将我们对窗口的设想和系统的窗口实现（通过 FLTK）连接起来的类。对于 Window 类实现的细节，可参考附录 E.3。

我们的按钮在 Simple_Window 的构造函数中被初始化：

```
Simple_window::Simple_window(Point xy, int w, int h, const string& title)
    :Window(xy,w,h,title),
    next_button(Point(x_max()-70,0), 70, 20, "Next", cb_next),
    button_pushed(false)
{
    attach(next_button);
}
```

不出意外的，Simple_window 将自己的位置(xy)，尺寸(w, h)和标题(title)传递给 Graph_lib 的 Window 类来处理。接下来，构造函数使用位置(Point(x_max()-70, 0)，大致位于右上角)、尺寸(70, 20)、标签(“Next”)和一个“回调”函数(cb_next)初始化 next_button。前四个参数的作用与我们对 Window 所做的相同：将一个矩形形状放在屏幕上并且为它添加标签。

最后，我们将 next_button 按钮添加到 Simple_window 中；也就是说，告知窗口必须要将这个按钮显示在它的位置上，并且保证 GUI 系统知道它的存在。

button_pushed 成员是一个相当隐晦的细节；我们用它来跟踪自从上次执行 next() 起到现在按钮是否被按下。事实上，这里做的每件事基本上都属于实现细节，因此将其声明为 private。忽略实现细节后的代码为：

```
struct Simple_window : Graph_lib::Window {
    Simple_window(Point xy, int w, int h, const string& title);

    void wait_for_button();    // simple event loop

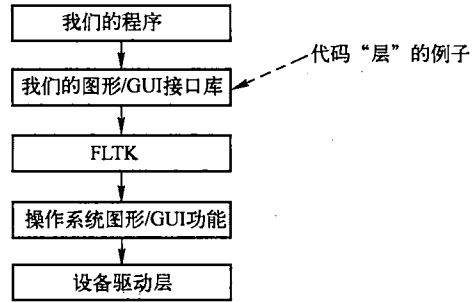
    //...
};
```


也就是说，用户可以创建一个窗口并等待按钮被按下。

16.3.1 回调函数

此处，函数 `cb_next()` 是一个新的，而且很有趣的代码。它就是当 GUI 系统检测到按钮被按下时，我们希望 GUI 系统调用的那个函数。由于我们将该函数交给 GUI，以便 GUI 能“回过头来调用我们”，所以它通常被称为“回调函数”。我们通过它的前缀 `cb_` 代表“回调”来指出 `cb_next()` 的用途。前缀的作用只是帮助我们理解——没有任何语言或者库有这样的命名要求。很明显，我们选择名字 `cb_next` 是因为它是“Next”按钮的回调函数。`cb_next` 的定义是一段比较丑陋的“样本”代码。

在给出代码之前，先让我们看看这里发生的事情，如右图所示。我们的程序是运行在多“层”代码之上的。它使用我们的图形库，而图形库是利用 FLTK 库实现的，FLTK 库又是使用操作系统功能实现的。在一个系统中，可能还会有更多的层和子层。无论以什么方式，当鼠标设备驱动检测到点击操作时，我们的函数 `cb_next()` 必须被调用。我们将 `cb_next()` 的地址和 `Simple_window` 对象的地址经过软件层次向下传递；当“Next”按钮被按下时，一些“下层”的代码就会调用 `cb_next()` 函数。



GUI 系统(和操作系统)可以被不同语言编写的程序使用，所以它不能向所有用户强加一些好的 C++ 风格。特别地，它并不知道我们的 `Simple_window` 类或者 `Button` 类。事实上，它根本就不知道类和成员函数的概念。回调函数的类型是经过小心选择的，以便可被低层的程序设计(包括 C 和汇编)所用。回调函数是没有返回值的，它接受两个地址参数。我们可以遵从这些规则来声明一个 C++ 成员函数如下：

```
static void cb_next(Address, Address); // callback for next_button
```

关键字 `static` 用于保证 `cb_next()` 可以作为一个普通函数被调用；也就是说，不是作为针对某个特定对象的 C++ 成员函数被调用。令系统能够正确调用一个 C++ 成员函数当然很好，但回调接口必须能被多种语言所用，所以我们只能将其定义为 `static` 类型的成员函数。`Address` 参数指明 `cb_next()` 的参数是“内存中某些内容”的地址。大部分语言都不支持 C++ 语言的引用，所以这里不能使用引用。编译器并不告诉你“那些内容”是什么类型的。在本例中，我们非常接近底层硬件，不能像往常那样从语言中得到帮助。“系统”激活一个回调函数时，传递给它的第一个参数是触发回调的 GUI 实体(Widget)的地址。本例中不需要使用第一个参数，所以我们不关心它的命名。第二个参数包含这个 Widget 的窗口的地址；对于 `cb_next()` 来说，它是我们的 `Simple_window` 对象。我们可以按照如下方式使用这个信息：

```
void Simple_window::cb_next(Address, Address pw)
// call Simple_window::next() for the window located at pw
{
    reference_to<Simple_window>(pw).next();
}
```

`reference_to<Simple_window>(pw)` 告诉编译器 `pw` 中的地址可以认为是 `Simple_window` 对象的地址；也就是说，我们可以将 `reference_to<Simple_window>(pw)` 当做 `Simple_window` 对象的引用来使用。在第 17、18 章中，我们将回到内存寻址的话题。在附录 E.1 中，我们给出了 `reference_to` 的定义。现在，我们只是乐于看到最后获得了一个指向 `Simple_window` 对象的引用，这样就可以像

以往那样访问我们的数据和函数。最后，通过调用我们的成员函数 `next()`，尽可能快地脱离和系统有关的代码。

我们本来可以将所有想要执行的代码都放在 `cb_next()` 中，但是像大多数好的 GUI 程序设计者一样，我们更愿意把这些麻烦的低层内容和精巧的用户代码相分离，所以使用下面两个函数来处理回调：

- `cb_next()` 简单地将回调函数的系统约定映射到一个普通的成员函数 (`next()`)。
- `next()` 实现我们实际要做的事情 (不需要知道回调函数的系统约定)。

使用两个函数的本质原因是一个通用设计原则，即“一个函数应该执行单一的逻辑行为”：`cb_next()` 使我们脱离了低层系统相关的部分，`next()` 执行我们期望的动作。任何时候，当我们需要一个 (来自“系统”) 对我们窗口的回调时，就可以定义这样一对函数；例如，可以参考 16.5 节 ~ 16.7 节。继续下一步之前，先重复一下到目前为止我们做了什么：

- 定义了 `Simple_window` 类。
- `Simple_window` 的构造函数将 `next_button` 对象注册到 GUI 系统。
- 当点击屏幕上的 `next_button` 时，GUI 调用 `cb_next()` 函数。
- `cb_next()` 将低层的系统信息转换成对窗口成员函数 `next()` 的调用。
- `next()` 执行我们想要做的事情以响应按钮点击的动作。

这是进行函数调用的一个相当完美的方法。不过要记住，我们是在处理鼠标 (或者其他硬件设备) 动作和程序之间的基本通信机制。特别地：

- 通常有很多程序同时在运行。
- 这些程序都是在操作系统之后很久才编写的。
- 这些程序都是在 GUI 库之后很久才编写的。
- 这些程序所用的语言可能和实现操作系统的语言完全不同。
- 这种技术处理所有类型的交互 (不仅仅是按下按钮这样的小例子)。
- 一个窗口可以有很多按钮；一个程序可以有很多窗口。

不过，一旦理解了 `next()` 是如何被调用的，我们就从本质上理解了如何使用 GUI 接口来处理程序中所有的动作。

16.3.2 等待循环

那么，在这种最简单的情况下，我们希望 `Simple_window` 的 `next()` 函数在每次按下按钮的时候做些什么呢？本质上，我们需要一个能将程序停止在某个点上的操作，以便有机会观察到目前为止已经完成了哪些工作。同时还希望 `next()` 函数能够从停止点重新启动程序：

```
// create some objects and/or manipulate some objects, display them in a window
win.wait_for_button(); // next() causes the program to proceed from here
// create some objects and/or manipulate some objects
```

实际上，这很容易做到。首先定义 `wait_for_button()`：

```
void Simple_window::wait_for_button()
// modified event loop:
// handle all events (as per default), quit when button_pushed becomes true
// this allows graphics without control inversion
{
    while (!button_pushed) Fl::wait();
    button_pushed = false;
    Fl::redraw();
}
```

像大多数 GUI 系统一样, FLTK 提供了一个停止程序运行的函数, 直到某个事件发生才重启程序。这个 FLTK 版本的函数称为 `wait()`。实际上, `wait()` 需要关注很多事情, 因为发生任何影响我们程序的事件都会将程序唤醒。例如, 如果程序运行在 Microsoft Windows 环境下, 当窗口被移动或者被其他窗口遮挡时, 窗口的重新绘制工作是由程序来完成的。Window 类还需要处理调整窗口尺寸的工作。在默认情况下, `Fl::wait()` 会处理所有这些工作。每当 `wait()` 函数处理完成某个事件后, 它总会返回, 使我们的代码有机会处理一些事情。

因此, 当有人点击“Next”按钮时, `wait()` 就会调用 `cb_next()` 并且返回(到我们的“等待循环”)。为了使 `wait_for_button()` 继续运行, `next()` 只需将布尔变量 `button_pushed` 设置为 `true`。这很容易:

```
void Simple_window::next()
{
    button_pushed = true;
}
```

当然, 还需要在某个合适的地方预先定义 `button_pushed`:

```
bool button_pushed = false;
```

等待之后, `wait_for_button()` 需要将 `button_pushed` 复位并且重绘 (`redraw()`) 窗口以保证我们所做的任何改变都能够在屏幕上表现出来, 这就是它所做的全部工作。

16.4 Button 和其他 Widget

定义如下按钮:

```
struct Button : Widget {
    Button(Point xy, int w, int h, const string& label, Callback cb);
    void attach(Window&);
};
```

由此可知, 按钮 (Button) 是一个具有位置 (xy)、尺寸 (w、h)、文本标签 (label) 和回调函数 (cb) 的 Widget。基本上, 任何出现在屏幕上, 带有关联动作 (例如回调函数) 的东西都是 Widget。

16.4.1 Widget

是的, 构件 (widget) 是一个技术术语。构件还有另一个名字——控件 (control), 虽然更具描述性, 但不够形象。我们使用构件来定义通过 GUI (图形用户界面) 与程序进行交互的形式。

Widget 接口类如下:

```
class Widget {
    // Widget is a handle to an Fl_widget — it is *not* an Fl_widget
    // we try to keep our interface classes at arm's length from FLTK
public:
    Widget(Point xy, int w, int h, const string& s, Callback cb);

    virtual void move(int dx, int dy);
    virtual void hide();
    virtual void show();
    virtual void attach(Window&) = 0;

    Point loc;
    int width;
    int height;
    string label;
    Callback do_it;
```

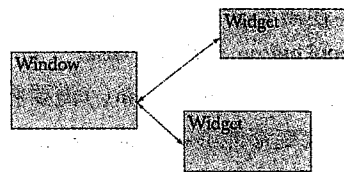
```
protected:
    Window* own;    // every Widget belongs to a Window
    Fl_Widget* pw;  // connection to the FLTK Widget
};
```

Widget 有两个有趣的函数，我们可以将它们用在 Button 上（以及很多其他从 Widget 派生出来的类，例如 Menu；参见 16.7 节）：

- hide() 使 Widget 对象不可见。
- show() 使 Widget 对象再次可见。

一个 Widget 对象开始是可见的。

如同 Shape 对象一样，我们可以在 Window 中移动(move())一个 Widget 对象，不过在进行该操作之前必须把它添加到 Window 对象中。注意，我们将 attach() 声明为一个纯虚函数（参见 14.3.5 节）：每一个派生自 Widget 的类必须定义它自己的 attach() 函数。事实上，系统级构件是在 attach() 函数中创建的。作为 Window 自己的 attach() 函数实现的一部分，构件的 attach() 函数是由 Window 对象调用的。实际上，连接窗口和构件如同跳双人舞蹈，各自必须完成自己的那部分工作。最终结果就是一个窗口知道它所包含的构件，而每个构件也知道它所属的窗口，如右图所示。注意，一个 Window 对象并不知道它所处理的 Widget 的类型。如 14.4 和 14.5 节中描述的那样，我们使用面向对象程序设计来保证 Window 可以处理每种类型的 Widget。同样，一个 Widget 对象也不知道它所属的 Window 的类型。



这个实现还不够完美，因为我们将数据成员定义为外部可访问的。而 own 和 pw 是严格用于派生类实现的，所以我们将它们声明为 protected。

在 GUI.h 中可以找到 Widget 类以及我们所使用的具体构件类(Button、Menu 等)的定义。

16.4.2 Button

Button 是一个最简单的 Widget，当我们点击按钮的时候，其功能只是调用一个回调函数：

```
class Button : public Widget {
public:
    Button(Point xy, int ww, int hh, const string& s, Callback cb)
        :Widget(xy,ww,hh,s,cb) {}

    void attach(Window& win);
};
```

这就是全部代码。attach() 函数包含所有（相对）繁琐的 FLTK 代码。我们将在附录 E 中进行相的介绍（在读完第 17、18 章之前不要去阅读附录 E）。现在，你只需要知道定义一个 Widget 并不是特别困难。

我们并不处理按钮（或其他 Widget）的外观，这个问题有些复杂和棘手。问题在于，对于外观风格我们几乎有无限多种选择，而有些风格是系统强制要求的。而且，从程序设计技术的角度来看，呈现不同的按钮外观并不需要任何新知识。如果这令你失望的话，你可以注意这样一个事实，将一个 Shape 对象放置在一个按钮上面，并不会对按钮的功能造成任何影响，而且你已经知道了如何生成任何需要的形状。

16.4.3 In_box 和 Out_box

我们提供了两种 Widget，用于文本输入/输出：

```

struct In_box : Widget {
    In_box(Point xy, int w, int h, const string& s)
        :Widget(xy,w,h,s,0) {}
    int get_int();
    string get_string();

    void attach(Window& win);
};

struct Out_box : Widget {
    Out_box(Point xy, int w, int h, const string& s)
        :Widget(xy,w,h,s,0) {}
    void put(int);
    void put(const string&);

    void attach(Window& win);
};

```

In_box 能够接受输入给它的文本，我们可以使用 `get_string()` 将文本作为字符串读出或者使用 `get_int()` 将其作为整数读出。如果你想知道是否已经有文本输入，可以使用 `get_string()` 读取并检查是否得到了空字符串：

```

string s = some_inbox.get_string();
if (s == "") {
    // deal with missing input
}

```

Out_box 用来向用户呈现信息。与 In_box 类似，我们可以使用 `put()` 输出字符串或者整数。16.5 节给出了使用 In_box 和 Out_box 的例子。

我们并没有提供 `get_floating_point()`、`get_complex()` 等函数。但不必为此担心，因为你可以获得字符串，并将它放入一个 `stringstream` 中，就可以按你喜欢的方式做任意的格式化输入了（参见 11.4 节）。

16.4.4 Menu

我们提供了一个非常简单的“菜单”的概念：

```

struct Menu : Widget {
    enum Kind { horizontal, vertical };
    Menu(Point xy, int w, int h, Kind kk, const string& label);
    Vector_ref<Button> selection;
    Kind k;
    int offset;
    int attach(Button& b);    // attach button to Menu
    int attach(Button* p);    // attach new button to Menu

    void show()               // show all buttons
    {
        for (unsigned int i = 0; i < selection.size(); ++i)
            selection[i].show();
    }
    void hide();              // hide all buttons
    void move(int dx, int dy); // move all buttons

    void attach(Window& win); // attach all buttons to Window win
};

```

Menu 本质上是一个按钮向量。跟以前一样，Point 对象 `xy` 指出左上角的位置。宽度和高度的作用是，当给菜单添加按钮的时候，用来重设按钮大小。参见 16.5 节和 16.7 节的例子。每一个菜单按钮（“菜单项”）是一个独立的 Widget，作为 `attach()` 的参数提供给 Menu。接着，Menu 提

供一个 `attach()` 操作将所有 `Button` 添加到 `Window` 对象。`Menu` 对象使用 `Vector_ref` (参见 13.10 节和附录 E.4) 跟踪它的所有的 `Button`。如果想要一个“弹出式”菜单, 你只能自己创建一个; 参见 16.7 节。

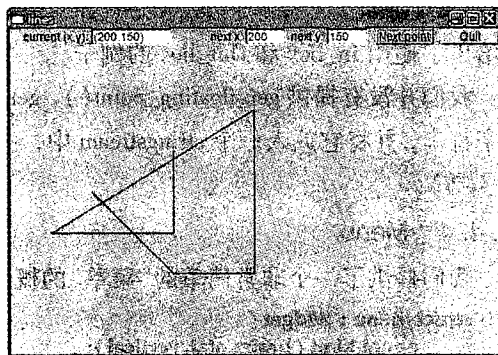
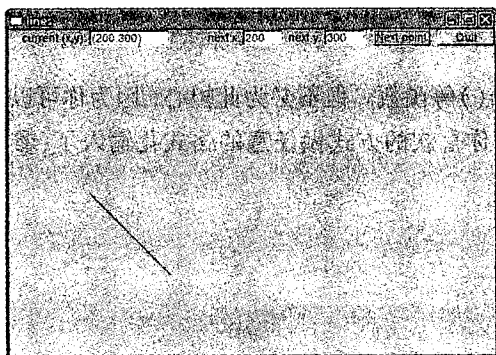
16.5 一个实例

为了更好地感受基本的 GUI 工具, 我们给出了一个简单的应用程序, 它是一个包含输入、输出和一些图形的窗口:

这个程序允许用户绘制一系列由坐标对指定的线段(开放多线段; 参见 13.6 节)。使用方法是用户反复在“next x”和“next y”框中输入(x, y)坐标对; 每输入一个坐标对就点击一次“next point”按钮。

开始时, “current(x, y)”框是空的, 程序等待用户输入第一个坐标对。用户输入后, 起点出现在“current(x, y)”框中, 每次输入的新坐标都会用来绘制一条线: 一条从当前点(显示在“current(x, y)”框中的坐标)到新输入点(x, y)之间的线, 然后(x, y)就成为新的当前点。

这样就能够绘制一条开放多线段, 完成之后用户可以通过“quit”按钮退出。整个过程非常直截了当, 同时该程序还展示了几个有用的 GUI 特性: 文本输入/输出、线的绘制和多按钮。上面的窗口显示了输入两个坐标对之后的结果; 输入 7 个坐标对则可得:



让我们来定义一个表示这种窗口的类。代码如下所示, 这段代码也很直接:

```
struct Lines_window : Window {
    Lines_window(Point xy, int w, int h, const string& title);
    Open_polyline lines;
private:
    Button next_button;    // add (next_x,next_y) to lines
    Button quit_button;
    In_box next_x;
    In_box next_y;
    Out_box xy_out;

    static void cb_next(Address, Address); // callback for next_button
    void next();
    static void cb_quit(Address, Address); // callback for quit_button
    void quit();
};
```

代码中使用 `Open_polyline` 对象表示线。代码还声明了按钮和文本框(分别为 `Button`、`In_box` 和 `Out_box`), 并且为每个按钮声明了一个实现其功能的成员函数和对应的回调函数。

Line_window 的构造函数负责初始化所有成员：

```
Lines_window::Lines_window(Point xy, int w, int h, const string& title)
: Window(xy,w,h,title),
  next_button(Point(x_max()-150,0), 70, 20, "Next point", cb_next),
  quit_button(Point(x_max()-70,0), 70, 20, "Quit", cb_quit),
  next_x(Point(x_max()-310,0), 50, 20, "next x:"),
  next_y(Point(x_max()-210,0), 50, 20, "next y:"),
  xy_out(Point(100,0), 100, 20, "current (x,y):")
{
    attach(next_button);
    attach(quit_button);
    attach(next_x);
    attach(next_y);
    attach(xy_out);
    attach(lines);
}
```

也就是说，创建每一个构件并把它们添加到窗口中。

对“Quit”按钮的处理是比较简单的：

```
void Lines_window::cb_quit(Address, Address pw) // "the usual"
{
    reference_to<Lines_window>(pw).quit();
}

void Lines_window::quit()
{
    hide(); // curious FLTK idiom for delete window
}
```

同以往一样：回调函数（这里是 `cb_quit()`）只是跳转到完成实际操作的函数（这里是 `quit()`）。这里的实际操作是删除窗口（Window），此处我们使用了 FLTK 的一个奇怪的特性——简单地隐藏窗口。

主要工作都在“Next point”按钮中完成。它的回调函数也很普通：

```
void Lines_window::cb_next(Address, Address pw) // "the usual"
{
    reference_to<Lines_window>(pw).next();
}
```

`next()` 函数定义了“Next point”按钮所做的实际工作：读取一个坐标对，更新 `Open_polyline` 对象，更新位置的输出，并且重绘窗口：

```
void Lines_window::next()
{
    int x = next_x.get_int();
    int y = next_y.get_int();

    lines.add(Point(x,y));

    // update current position readout:
    stringstream ss;
    ss << '(' << x << ', ' << y << ')';
    xy_out.put(ss.str());

    redraw();
}
```

这段代码很容易理解。我们用 `get_int()` 从 `In_box` 得到整数坐标值；用一个 `stringstream` 对象来格式化要输出到 `Out_box` 的字符串；`str()` 成员函数负责从 `stringstream` 对象中读取字符串；`redraw()`

函数将最终结果显示给用户。注意，直到 Window 的 `redraw()` 函数被调用之前，屏幕上一直显示的是旧图像。

那么，这个程序有什么奇怪和不同之处呢？让我们看看它的 `main()` 函数：

```
#include "GUI.h"

int main()
try {
    Lines_window win(Point(100,100),600,400,"lines");
    return gui_main();
}
catch(exception& e) {
    cerr << "exception: " << e.what() << "\n";
    return 1;
}
catch (...) {
    cerr << "Some exception\n";
    return 2;
}
```

这里基本没有做任何事情！`main()` 函数体只是定义了窗口 `win` 并调用函数 `gui_main()`。这里并没有其他函数，没有任何在第 6、7 章介绍过的 `if`、`switch` 或循环等代码，仅仅是一个变量的定义和对函数 `gui_main()` 的调用，而 `gui_main()` 本身也只是调用 FLTK 的 `run()` 函数。更进一步，我们会发现 `run()` 函数也只是一个简单的无限循环：

```
while(wait());
```

除了少数几个将在附录 E 中介绍的实现细节外，我们已经看到了令画线程序运行起来的所有代码和所有的基本逻辑。那么，前面发现的奇怪问题到底是怎么回事呢？

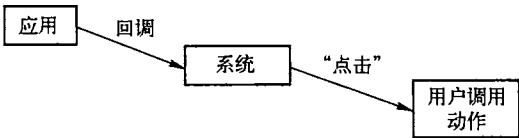
16.6 控制流的反转

这里的关键就是，我们将执行序的控制权从程序交给了构件：无论用户激活、运行哪个构件，都会发生这种情况。例如，点击一个按钮就会运行它的回调函数。当回调函数返回以后，程序就挂起，等待用户进行其他处理。本质上，`wait()` 告诉“系统”关注这些构件并调用对应的回调函数。理论上，`wait()` 可以告诉程序员哪个构件要求这种关注，并将调用恰当函数的任务留给程序员来做。然而，在 FLTK 和其他大多数 GUI 系统中，`wait()` 只是简单地调用适当的回调函数，从而免去了程序员编写相应代码的麻烦。

一个“常规程序”的结构为：



一个“GUI 程序”的结构为：



“控制流反转”的一个含义是程序的执行顺序完全由用户的行为决定。这使得程序的组织和调试都更加复杂。很难猜想用户想要做什么，也很难想象一个随机回调序列可能产生的所有影响。这使得系统测试非常困难(参见第 26 章)。处理这些问题的技术已经超出了本书的讨论范围，但是我们建议你格外小心那些由用户通过回调来驱动的代码。除了明显的控制流问题之外，还有关于可见性的问题，以及跟踪构件与数据关联的困难。为了尽量减少麻烦，关键是要保证 GUI 部分的程序简洁性，同时逐步增量式构建一个 GUI 程序，并且在每一个阶段都要测试。当你编写一个 GUI 程序的时候，绘制对象及它们之间交互的图表也是很关键的。

被不同回调函数触发的代码是如何相互通信的呢？最简单的方法是处理保存在窗口中的数据，就像 16.5 节中的例子那样。在那个例子中，Lines_window 的 next() 函数通过点击“Next point”按钮被调用，它从 In_box 读取数据(next_x 和 next_y)，并且更新 lines 成员变量和 Out_box(xy_out)。显然，由回调调用的函数可以做任何事情：可以打开文件、连接网络等。但是，目前我们只考虑简单的情况：将数据保存在窗口中。

16.7 添加菜单

我们下面通过为画线程序添加菜单，来进一步研究“控制流反转”带来的控制和通信问题。首先，我们提供一个简单的菜单，允许用户改变 lines 成员变量中所有线的颜色。下面我们添加 color_menu 菜单及其回调函数：

```
struct Lines_window : Window {
    Lines_window(Point xy, int w, int h, const string& title);

    Open_polyline lines;
    Menu color_menu;

    static void cb_red(Address, Address);    // callback for red button
    static void cb_blue(Address, Address);   // callback for blue button
    static void cb_black(Address, Address);  // callback for black button

    // the actions:
    void red_pressed() { change(Color::red); }
    void blue_pressed() { change(Color::blue); }
    void black_pressed() { change(Color::black); }

    void change(Color c) { lines.set_color(c); }

    // ... as before ...
};
```

重复写出这些几乎相同的回调函数和“动作”函数非常乏味。但是，这在概念上比较简单，而且那些在输入方面更加简单的方式也超出了本书的讨论范围。当一个菜单按钮被按下时，它会将线改为所要求的颜色。

我们需要初始化已经定义的 color_menu 成员：

```
Lines_window::Lines_window(Point xy, int w, int h, const string& title)
    : Window(xy, w, h, title),
    // ... as before ...
    color_menu(Point(x_max()-70,40),70,20,Menu::vertical,"color")
{
    // ... as before ...
    color_menu.attach(new Button(Point(0,0),0,0,"red",cb_red));
    color_menu.attach(new Button(Point(0,0),0,0,"blue",cb_blue));
}
```

```

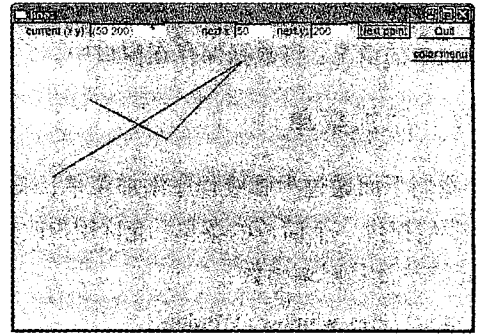
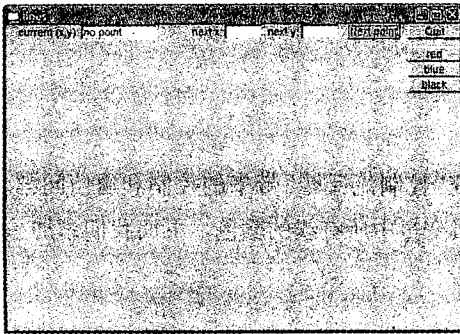
    color_menu.attach(new Button(Point(0,0),0,0,"black",cb_black));
    attach(color_menu);
}

```

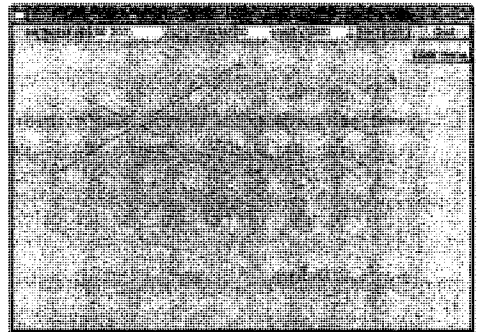
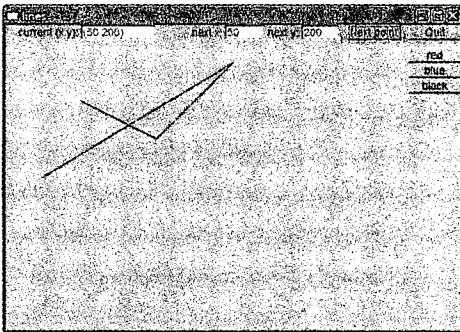
这些按钮是动态添加到菜单上的(使用 `attach()`), 并且可以根据需要移除和替换它们。`Menu::attach()` 调整按钮的尺寸和位置, 并将它们添加到窗口中。这就是这段代码的全部工作, 其运行结果如下:

程序运行了一段时间之后, 我们发现真正需要的是一个“弹出式”菜单; 也就是说, 我们不希望把稀有的屏幕空间用在一个菜单上, 除非我们正在使用它。因此, 我们添加一个“color menu”按钮。当按下它的时候, 弹出颜色菜单, 并且在完成一个选择操作之后, 菜单重新隐藏起来, 而“color menu”按钮再次出现在窗口中。

添加了几条线之后的窗口如下:



我们看到了新的“color menu”按钮和一些(黑色的)线。点击“color menu”按钮会出现颜色菜单: 注意, 这时候“color menu”按钮隐藏了, 在使用颜色菜单完成操作之前并不需要这个按钮。点击“blue”按钮后可得:



现在, 线都变成了蓝色并且“color menu”按钮重新出现在窗口中。

为了达到这种效果, 我们添加了“color menu”按钮并且修改那些“点击”函数来调整菜单和按钮的可见性。下面是 `Lines_window` 的完整实现:

```

struct Lines_window : Window {
    Lines_window(Point xy, int w, int h, const string& title);
private:
    // data:
    Open_polyline lines;

    // widgets:
    Button next_button; // add (next_x,next_y) to lines

```

```

Button quit_button; // end program
In_box next_x;
In_box next_y;
Out_box xy_out;
Menu color_menu;
Button menu_button;
void change(Color c) { lines.set_color(c); }

void hide_menu() { color_menu.hide(); menu_button.show(); }
// actions invoked by callbacks:
void red_pressed() { change(Color::red); hide_menu(); }
void blue_pressed() { change(Color::blue); hide_menu(); }
void black_pressed() { change(Color::black); hide_menu(); }
void menu_pressed() { menu_button.hide(); color_menu.show(); }
void next();
void quit();

// callback functions:
static void cb_red(Address, Address);
static void cb_blue(Address, Address);
static void cb_black(Address, Address);
static void cb_menu(Address, Address);
static void cb_next(Address, Address);
static void cb_quit(Address, Address);
};

```

注意，除了构造函数以外，其他成员都是私有的。本质上，这个窗口类就是完整程序。所有工作都是通过它的回调函数完成的，因此不需要窗口之外的任何代码。我们将声明进行了简单排列，以使类更可读。构造函数提供了所有子对象的参数并将这些对象添加到窗口中：

```

Lines_window::Lines_window(Point xy, int w, int h, const string& title)
: Window(xy, w, h, title),
  next_button(Point(x_max()-150,0), 70, 20, "Next point", cb_next),
  quit_button(Point(x_max()-70,0), 70, 20, "Quit", cb_quit),
  next_x(Point(x_max()-310,0), 50, 20, "next x:"),
  next_y(Point(x_max()-210,0), 50, 20, "next y:"),
  xy_out(Point(100,0), 100, 20, "current (x,y):"),
  color_menu(Point(x_max()-70,30), 70, 20, Menu::vertical, "color"),
  menu_button(Point(x_max()-80,30), 80, 20, "color menu", cb_menu),
{
  attach(next_button);
  attach(quit_button);
  attach(next_x);
  attach(next_y);
  attach(xy_out);
  xy_out.put("no point");
  color_menu.attach(new Button(Point(0,0), 0, 0, "red", cb_red));
  color_menu.attach(new Button(Point(0,0), 0, 0, "blue", cb_blue));
  color_menu.attach(new Button(Point(0,0), 0, 0, "black", cb_black));
  attach(color_menu);
  color_menu.hide();
  attach(menu_button);
  attach(lines);
}

```

注意，初始化的顺序和数据成员的声明顺序是一致的，这是书写初始化代码的正确顺序。事实上，成员初始化的执行顺序总是按照成员声明的顺序。如果一个基类或成员的构造函数的次序不对，一些编译器会给出警告信息。

16.8 调试 GUI 代码

一旦 GUI 程序开始工作，通常就很容易调试了：因为你看到的就是你得到的。然而，在第一个形状和构件开始出现在窗口中，甚至是窗口出现在屏幕上之前，通常会是一个充满挫折的阶段。试试下面的 `main()` 函数：

```
int main()
{
    Lines_window(Point(100,100),600,400,"lines");
    return gui_main();
}
```

你看到错误了吗？无论你是否看到了，你都应该试一试；程序可以编译通过并运行，但是 `Lines_window` 并未给你画线的机会，你能得到的最多是屏幕上的一闪而已。如何从这样的程序中找到错误呢？

- 小心使用经过严格验证的程序组件(类、函数、库)。
- 简化所有的新代码，降低程序从最简版本“增长”的速度，仔细逐行检查代码。
- 检查所有的链接设置。
- 与已经正常运行的程序比较。
- 向朋友解释代码。

你会发现很难跟踪代码的执行过程。如果你已经学会了使用调试器，你可能还有机会，但在这种情况下，只是加入“输出语句”将不再有效——因为根本看不到任何输出。即使是使用调试器也有问题，因为有很多程序在同时运行(“多线程”)——你的代码并不是唯一试图和屏幕交互的代码。代码简化和理解代码的系统方法是关键。

那么前面 `main()` 函数的问题出在哪儿呢？下面给出了正确的版本(来自 16.5 节)：

```
int main()
{
    Lines_window win(Point(100,100),600,400,"lines");
    return gui_main();
}
```

我们“忘记”了 `Lines_window` 的名字 `win`。因为我们实际上并不需要这个名字，这看起来是合理的，但是编译器会认为既然我们不再使用这个窗口，那就要立即销毁它。天啊！这个窗口的生命周期仅是毫秒级，这样出现前面的结果就不足为奇了。

另一个常见的问题是将一个窗口恰好放在了另一个窗口上面，明显(或者不是非常明显)看起来就像只有一个窗口。另一个窗口到哪儿去了呢？我们很可能会寻找这种代码中并不存在的错误，而浪费宝贵的时间。如果我们把一个形状放在了另一个形状上面，也可能出现同样的问题。

还有，可能会让事情更糟的是，当使用 GUI 库的时候，异常并不总是如我们希望的那样正常工作。因为代码由 GUI 库控制，我们抛出的异常可能永远不会到达我们的处理程序，GUI 库或者操作系统可能会“吃掉”它(即它们可能依赖不同于 C++ 异常的错误处理机制，可能完全忽略了 C++)。

在调试中经常发现的问题包括：由于 `Shape` 和 `Widget` 对象没有被添加到窗口中而没有显示；由于超出对象的作用域导致出错。考虑程序员如何在菜单中创建并添加按钮：

```
// helper function for loading buttons into a menu
void load_disaster_menu(Menu& m)
{
    Point orig(0,0);
    Button b1(orig,0,0,"flood",cb_flood);
    Button b2(orig,0,0,"fire",cb_fire);
    // ...
    m.attach(b1);
    m.attach(b2);
    // ...
}

int main()
{
    // ...
    Menu disasters(Point(100,100),60,20,Menu::horizontal,"disasters");
    load_disaster_menu(disasters);
    win.attach(disasters);
    // ...
}
```

上面的代码不能正常运行。所有按钮都是 `load_disaster_menu()` 函数内的局部变量，将它们添加到菜单上并不会改变这一点。在 18.5.4 节中可以找到这一问题的解释（不要返回指向局部变量的指针），而 8.5.8 节说明了局部变量的内存布局情况。这里的关键是，在 `load_disaster_menu()` 函数返回之后，那些局部对象就已经被销毁了，`disasters` 菜单将指向不存在（销毁了）的对象。结果肯定是错误的并且令人吃惊的。解决方法是使用 `new` 创建的未命名对象而不是命名的局部对象：

```
// helper function for loading buttons into a menu
void load_disaster_menu(Menu& m)
{
    Point orig(0,0);
    m.attach(new Button(orig,0,0,"flood",cb_flood));
    m.attach(new Button(orig,0,0,"fire",cb_fire));
    // ...
}
```

正确方法甚至比（十分常见的）错误方法更加简单。

简单练习

1. 使用 FLTK 的链接程序设置（如附录 D 中的描述）并建立一个新项目。
2. 使用 `Graph_lib` 的工具，输入 16.5 节的画线程序并运行它。
3. 使用 16.7 节中描述的那种“弹出式”菜单修改程序并运行它。
4. 再次修改这个程序，添加第二个菜单用于选择线型，并运行它。

思考题

1. 为什么需要图形用户界面？
2. 什么时候需要非图形用户界面？
3. 什么是软件的层次结构？
4. 为什么需要对软件分层？
5. C++ 程序与操作系统通信时的基本问题是什么？
6. 什么是回调函数？
7. 什么是构件？

8. 构件的另一个名称是什么?
9. 首字母缩写词 FLTK 是什么意思?
10. FLTK 如何发音?
11. 你还听说过其他的 GUI 工具集吗?
12. 哪些系统使用术语构件? 哪些系统更喜欢使用控件一词?
13. 构件的例子有哪些?
14. 什么时候需要使用输入框?
15. 输入框中的值是什么类型的?
16. 什么时候需要使用按钮?
17. 什么时候需要使用菜单?
18. 什么是控制流反转?
19. 调试 GUI 程序的基本策略是什么?
20. 为什么调试 GUI 程序比调试“普通的流式输入/输出程序”更难?



术语

按钮	对话框	可见/隐藏	回调	GUI	等待输入
控制台 I/O	菜单	等待循环	控制流	软件层次	构件
控制流反转	用户接口				



习题

1. 创建一个 `My_window` 类, 它比 `Simple_window` 多两个按钮——`next` 和 `quit`。
2. 创建一个带有 4×4 的正方形按钮方阵的窗口(基于 `My_window`)。当按下按钮时执行一个简单的动作, 比如在一个输出框中打印它的坐标, 或者变换为一个稍微不同的颜色(直到按下另一个按钮)。
3. 在按钮(`Button`)上放置一个图像(`Image`), 当按下按钮时移动按钮和图像。使用下面的随机数发生器为“图像按钮”选择新位置:

```
int rint(int low, int high) { return low+rand()%(high-low); }
```

 它返回一个 $[low, high)$ 内的随机整数。
4. 创建一个菜单, 包括绘制圆、正方形、等边三角形、六边形等菜单项。创建一个(或两个)输入框用于输入坐标对, 将点击某个菜单项后绘制的形状放置在这个坐标指定的位置上。抱歉, 没有拖放功能。
5. 编写程序绘制一个你选择的形状, 并在每次点击“Next”时将其移动到一个新的位置。这个新位置根据从输入流读取的一个坐标对决定。
6. 编写一个“模拟时钟”, 即一个带有转动指针的时钟。通过一个库函数调用从操作系统获取时间值。这个练习的主要工作是找到能够获取时间的函数, 找到等待一小段时间(比如, 1 秒)的方法, 以及根据你找到的文档学会使用它们。提示: 可考虑使用 `clock()`、`sleep()` 函数。
7. 使用前面练习中的技术, 创建一个飞机图像并让它在窗口中“飞来飞去”。窗口有一个“start”按钮和一个“stop”按钮。
8. 编写一个货币换算器。在启动时从一个文件读取汇率。在输入窗口中输入数额, 然后提供一种选择需换算的两种货币类型的方式(例如, 使用两个菜单)。
9. 修改第 7 章的计算器程序, 令它从输入框获取输入, 并将结果返回到输出框。
10. 编写一个程序, 可以让用户从一组数学函数(例如 `sin()` 和 `log()`)中进行选择, 并为这些函数提供参数, 然后将它们图形化显示出来。



附言

GUI 是一个庞大的主题, 它的很多内容与已有系统的风格和兼容性有关, 而且必须处理种类过于繁多的构件(例如 GUI 库提供许多不同的按钮风格), 可能植物学家对此会感到更亲切些。然而, 其中只有很少一部分与重要的程序设计技术有关, 所以我们不需要在这方面进行研究。其他的主题如按比例缩放、旋转、变

形、三维对象、阴影等，涉及很多图形学和数学方面的话题我们在本章中并未讨论。

你必须要知道的一件事情是，多数 GUI 系统都提供了一个“GUI 程序生成工具”，你可以在图形方式下设计你的窗口布局，将回调和动作函数与按钮、菜单等关联起来。对许多应用程序来说，这样一个 GUI 生成工具有助于减少那些编写“框架式代码”（例如我们的回调函数）的乏味工作。然而，我们应该尝试理解程序是如何运行的。有时，这类工具生成的代码与你本章中看到的代码相同。有时，这类代码可能会使用一些更加精致和/或代价更高的机制。

数据库系统是由数据库、数据库管理系统、数据库管理员、用户等组成的。数据库是存储在计算机中、有组织的数据集合。数据库管理系统是位于用户与操作系统之间的一层数据管理软件。数据库管理员是负责数据库系统运行、维护和管理的人员。用户是数据库系统的最终使用者。数据库系统的主要功能包括：数据定义、数据操纵、数据控制、数据查询、数据维护等。

第三部分 数据结构和算法

第 17 章 向量和自由空间

“默认使用向量!”

——Alex Stepanov

本章和后面四章将介绍 C++ 标准库(通常称为 STL)的容器和算法部分。我们介绍 STL 的关键功能和一些用途。另外,我们介绍用于实现 STL 的关键设计和编程技术,以及它的一些低层语言特点。它们主要是指针、数组和自由空间。本章和后面两章的重点是最通用和最有用的 STL 容器:向量的设计和实现。

17.1 介绍

C++ 标准库中最有用的容器是向量。一个向量提供一系列指定类型的元素。你可以通过它的索引(下标)找到一个元素,使用 `push_back()` 来扩展向量,使用 `size()` 来获得一个向量中的元素数量,以及防止对超出范围的向量元素的访问。标准库向量是一个方便的、灵活的、有效的(在时间和空间上)、静态的、类型安全的元素容器。标准字符串具有相似的属性,另外还有其他有用的容器类型,例如列表和映射,我们将在第 20 章中介绍。但是,一台计算机的内存并不能直接支持这些有用的类型。硬件直接支持的只是字节序列。例如,对于一个 `vector < double >`, 操作 `v.push_back(2.3)` 将 2.3 添加到 `double` 类型序列中,并将元素数量 `v(v.size())` 增加 1。在最低的层次中,计算机并不知道任何像 `push_back()` 那样复杂的事情;它所知道的只是如何一次读或写多个字节。

在本章和后面两章中,我们将展示如何通过基本语言功能来创建向量,这对于每个程序员都是有用的。这样,我们可以说明这些有用的概念和编程技术,而且还可以通过 C++ 语言的特点来把它们表示出来。我们在向量的实现中遇到的语言功能和编程技术,通常是有用的和被广泛使用的。

在学习如何设计、实现和使用向量之后,我们可以开始研究其他的标准库容器(例如映射),并测试 C++ 标准库所提供的简洁的、有效的功能(参见第 20 和 21 章)。这些功能称为算法,它可以使我们从涉及数据的一般任务中解脱出来。实际上,每一个 C++ 的实现都提供了许多算法为我们使用,这可以大大简化编写和测试我们自己的库的工作。我们曾经看过和使用过标准库中的一个最有用的算法: `sort()`。

我们通过一系列越来越复杂的向量实现来了解标准库。首先，我们创建一个非常简单的向量。然后，我们看向量不希望得到什么并修改它。这样反复几次之后，我们得到一个与标准库向量大致相当的向量，也就是你的 C++ 编译器所带的库，在前面章节中已经使用过。这种逐渐完善的过程与我们接触一个新的编程任务的方式非常相似。在此过程中，我们会遇到和讨论涉及内存和数据结构使用的很多经典问题。基本计划是：

- 第 17 章(本章)：如何处理大小不同的内存？特别地，不同向量如何拥有不同数量的元素，一个向量如何在不同时间拥有不同数量的元素？这促使我们使用自由空间(堆空间)、指针、转换(显式的类型转换)和引用。
- 第 18 章：如何复制向量？如何为它们提供下标操作？我们也会介绍数组，并讨论它与指针的关系。
- 第 19 章：如何使向量具有不同的元素类型？如何处理越界错误？为了回答这个问题，我们将讨论 C++ 模板和异常处理功能。

除了在实现灵活的、有效的、类型安全的向量时介绍新的语言功能和技术之外，我们也将使用我们曾经提到过的语言功能和编程技术。偶尔，我们也有机会给出一些更加正式和技术上的定义。

好了，现在是我们开始直接处理内存的时候了。为什么我们要这样做？向量和字符串是非常有用和方便的，我们可以仅仅使用它们。毕竟，这些容器(例如向量和字符串)设计目的之一就是将我们与实际处理内存时的某些不愉快因素隔离。但是，除非相信魔法，否则我们必须进行最低层次的内存管理。为什么你不能“只相信魔法”？(或者对它抱一个更积极的态度)为什么你不能“相信向量的实现者知道他们在做什么”？毕竟，我们不建议你了解使计算机的内存运转起来的物理设备。

原因是，我们是程序员(计算机科学家、软件开发者或其他人员)而不是物理学家。如果我们正在学习器件物理学，我们可能会要查看计算机内存的设计细节。但是，既然我们正在学习程序设计，就必须深入程序设计的细节。从理论上来说，我们可以像对库物理设备那样来看待低层次的内存访问和管理功能的“实现细节”。但是，如果我们那样去做，你将不能只是“相信魔法”；你将不能实现一个新的容器(会有这种需求的，这不罕见)。另外，你将不能阅读大量的直接使用内存的 C 与 C++ 代码。当我们阅读后面几章时，指针(一种低层次的、直接指向一个对象的方式)对很多与内存管理无关的问题也是有用的。在不使用指针的情况下，使用好 C++ 并不是容易的事。

更有哲理的是，我和很多计算机专业人士持同样的观点，如果你对一个程序如何映射到计算机内存和操作缺乏基本和实际的理解，你将会在把握高层次的主题时遇到问题，例如数据结构、算法和操作系统。

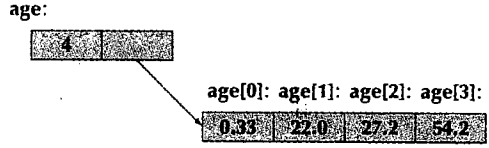
17.2 向量的基本知识

我们开始对向量的循序渐进的设计，考虑一个非常简单的用途：

```
vector<double> age(4);    // a vector with 4 elements of type double
age[0]=0.33;
age[1]=22.0;
age[2]=27.2;
age[3]=54.2;
```

很明显，我们创建一个有四个 double 类型元素的向量，并且给这四个元素分别赋值为 0.33、22.0、27.2 和 54.2。将这四个元素编号为 0、1、2 和 3。对 C++ 标准库容器中的元素编号只能从 0 开

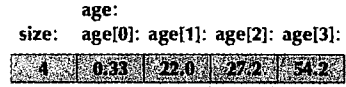
始。从 0 开始编号是很常用的，它是 C++ 程序员之间的普遍约定。一个向量中的元素数量称为它的大小。因此，age 的大小为 4。一个向量中的元素被编号(索引)为 0 到 size - 1。例如，age 中的元素被编号为 0 到 age.size() - 1。我们可以用右图来表示 age。如何将这种“图形化设计”用于实际的计算机内存中？如何获得像这样存储和访问的值？很明显，我们需要定义一个类，并且将这个类称为 vector。另外，它需要一个数据成员保存它的大小，以及另一个数据成员保存它的元素。但是，我们如何来表示一组数量可以变化的元素？可以使用一个标准库向量，但是(按照上下文)可能会被欺骗：因为我们要创建向量。



因此，我们如何表示上图中的箭头？考虑如何不使用它来工作。我们可以定义一个固定大小的数据结构：

```
class vector {
    int size, age0, age1, age2, age3;
    // ...
};
```

忽略符号方面的细节，我们将得到右图。这个过程是简单和方便



的，但是我们第一次用 push_back() 试图添加一个元素时就遇到了这样的问题：我们无法增加一个元素，程序中的元素数量固定为 4 个。如果我们将向量定义为元素数量固定，那么改变元素数量的操作(例如 push_back())就不能实现。基本上，我们需要一个数据成员来指向一组元素，这样当我们需要更大空间时可以使它指向不同组的元素。我们需要像第一个元素的内存地址这样的内容。在 C++ 中，一种可以保存地址的数据类型称为指针，它在语法上使用后缀* 来区分，因此 double* 是指“指向 double 的指针”。鉴于此，我们可以定义自己的第一个版本的 vector 类：

```
// a very simplified vector of doubles (like vector<double>)
class vector {
    int sz; // the size
    double* elem; // pointer to the first element (of type double)
public:
    vector(int s); // constructor: allocate s doubles,
                  // let elem point to them
                  // store s in sz
    int size() const { return sz; } // the current size
};
```

在我们进行向量的设计之前，我们首先详细学习“指针”的概念。“指针”和与它紧密相关的“数组”的概念，都是 C++ 中的“内存”概念的关键部分。

17.3 内存、地址和指针

一台计算机的内存是一个字节的序列。我们可以将这些字节从 0 到最后一个编号。我们将这种“一个指出在内存中位置的数字”称为地址。我们可以将一个地址看做是一种整数值。内存中第一个字节的地址为 0，下一个字节的地址为 1，以此类推。我们可以将 1M 字节可视化为以下图形：



我们在内存中保存的任何东西都有一个地址。例如：

```
int var = 17;
```

我们将为 `var` 分配一段“int 大小”的内存，并将值 17 保存在这段内存中。我们也可以保存和操作地址。一个保存地址的对象称为指针。例如，用于保存一个 int 的地址类型称为一个“指向 int 的指针”或“int 指针”，并且它的表示方法为 `int*`：

```
int* ptr = &var;           // ptr holds the address of var
```

“地址”操作符 `&` 用于获得一个对象的地址。因此，如果 `var` 碰巧开始于地址 4096 (或 2^{12})，`ptr` 将会保存值 4096：



基本上，我们将计算机内存看做是一个字节的序列，它们被编号为从 0 到内存大小减 1。对于有些计算机，这是一个简化模型，但是作为一种初级的编程模型，这已经足够了。

每种类型都有对应的指针类型。例如：

```
char ch = 'c';  
char* pc = &ch;           // pointer to char
```

```
int ii = 17;  
int* pi = &ii;           // pointer to int
```

如果我们想看到指针指向的对象值，我们可以使用“内容”操作符 `*`，例如：

```
cout << "pc==" << pc << "; contents of pc==" << *pc << "\n";  
cout << "pi==" << pi << "; contents of pi==" << *pi << "\n";
```

`*pc` 的输出将是字符 `c`，而 `*pi` 的输出将是整数 17。`pc` 和 `pi` 的输出依赖于编译器在内存中分配给变量 `ch` 和 `ii` 的地址。指针值 (地址) 的表示方法也可能依赖于你的系统遵守的规范；十六进制表示法 (参见附录 A.2.1.1) 常用于指针值。

操作符的内容 (经常称为解引用 (dereference) 操作符) 也可以用于赋值操作的左侧：

```
*pc = 'x';           // OK: you can assign 'x' to the char pointed to by pc  
*pi = 27;           // OK: an int* points to an int so *pi is an int  
*pi = *pc;           // OK: you can assign a char (pc) to an int (pi)
```

需要注意的是，即使指针的值可以被打印为一个整数，但是指针并不是一个整数。“一个 int 指向什么”不是一个好的问题；`int` 并不指向什么，而指针会这样做。指针类型提供适当的地址操作，`int` 会提供适于整数的 (算术和逻辑) 操作。因此，指针和整数不能混淆：

```
int i = pi;           // error: can't assign an int* to an int  
pi = 7;               // error: can't assign an int to an int*
```

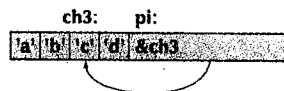
与此类似，一个指向 `char` 的指针 (`char*`) 不是一个指向 `int` 的指针 (`int*`)。例如：

```
pc = pi;             // error: can't assign an int* to a char*  
pi = pc;             // error: can't assign a char* to an int*
```

为什么将 `pc` 赋值给 `pi` 会出现错误？考虑一个答案：一个 `char` 通常比一个 `int` 更小，因此我们可以这样考虑：

```
char ch1 = 'a';  
char ch2 = 'b';  
char ch3 = 'c';  
char ch4 = 'd';  
int* pi = &ch3;       // point to ch, a char-sized piece of memory  
  
                        // error: we cannot assign a char* to an int*  
                        // but let's pretend we could  
*pi = 12345;           // write to an int-sized piece of memory  
*pi = 67890;
```

编译器如何在内存中分配变量是由实现定义的，但是我们很可能见到像下图这样的分配情况。现在，如果编译器对这段代码没有异议，我们可以将 12345 写入从 &ch3 开始的内存。这个过程可能会改变相邻内存的值，例如 ch2 或 ch4。如果我们确实不幸运（很容易发生），我们将会覆盖



pi 本身的部分！在那种情况下，下次赋值 `*pi = 67890` 会将 67890 放入内存中完全不同的部分。令人高兴的是这种赋值是不允许的，这是编译器为低层次编程提供的少数几种保护功能。

在不太可能出现的情况下，可能你需要把一个 `int` 转换成一个指针，或者将一个指针类型转换成其他类型，你将需要使用 `reinterpret_cast`，参见 17.8 节。

我们在这里确实需要接近硬件。这对于一个程序员来说，并不是一个特别舒服的地方。我们只有很少几种可用的原始操作，并且几乎不能得到语言或标准库的支持。但是，我们不得不了解高层功能（例如向量）如何实现。我们需要理解如何在这个层次编写代码，这是由于并不是所有代码位于“高层”（见第 25 章）。同样，我们更欣赏软件高层的方便和相对的安全，实际上我们已经体验过缺乏这些特性会带来什么后果。我们的目标是，对于给定的问题及求解方案的限制永远尽量在最高抽象层工作。在本章和第 18、19 章中，我们通过向量的实现来介绍如何回到抽象的更舒服的层次。

17.3.1 运算符 sizeof

那么，一个 `int` 实际占用多少内存？一个指针？操作符 `sizeof` 会回答这些问题：

```
cout << "the size of char is " << sizeof(char) << ' ' << sizeof('a') << '\n';
cout << "the size of int is " << sizeof(int) << ' ' << sizeof(2+2) << '\n';
int* p = 0;
cout << "the size of int* is " << sizeof(int*) << ' ' << sizeof(p) << '\n';
```

正如你所看到的，我们可以将 `sizeof` 用于一个类型名或表达式。对于一个类型名来说，`sizeof` 给出这种类型对象的大小；对于一个表达式来说，`sizeof` 给出表达式结果的大小。`sizeof` 的结果是一个正整数，`sizeof(char)` 单元的大小被定义为 1。在典型情况下，一个 `char` 被保存在一个字节中，因此 `sizeof` 会报告占用的字节数。

试一试 执行上面这个例子，并看我们能得到什么。然后，扩展这个例子以决定 `bool`、`double` 和其他类型的大小。

每个 C++ 实现并不保证一种类型的大小相同。目前，`sizeof(int)` 在台式机或笔记本中通常为 4 个字节。如果使用 8 比特的字节，那就意味着一个 `int` 是 32 比特。但是，嵌入式系统通常使用 16 比特的 `int` 型，而高性能体系结构使用 64 比特的 `int` 型。

一个向量使用多少内存？我们可以尝试如下：

```
vector<int> v(1000);
cout << "the size of vector<int>(1000) is " << sizeof(v) << '\n';
```

这个输出将会像这样：

```
the size of vector<int>(1000) is 20
```

看过本章和第 18 章（以及 19.2.1 节）的介绍后，这个解释将会变得很明显，但是 `sizeof` 明显不能用于统计向量元素数量。

17.4 自由空间和指针

思考一下 17.2 节结尾的向量实现。向量从哪里获得它的元素的空间？我们如何使指针 `elem` 指向它们？当你开始一个 C++ 程序时，编译器为你的代码分配内存（有时称为代码存储或文本存

储), 以及为你定义的全局变量分配内存(称为静态存储)。当你需要调用函数或你的参数和本地变量需要空间(称为堆栈存储或自动存储)时, 编译器也会为它分配内存分配: 一些内存。计算机中的其他内存可用于其他应用; 它是“空闲的”。我们可以用右图来说明。C++ 语言用称为 new 的操作符将“空闲存储”(又称为堆)变为可用状态。例如:

```
double* p = new double[4]; // allocate 4 doubles on the free store
```

这就要求 C++ 运行时系统在空闲空间中分配 4 个 double, 并且为我们返回一个指向第一个 double 的指针。我们使用得到的指针来初始化指针变量 p。我们可以用下图来表示:

new 操作符返回一个指向它创建的对象指针。如果它创建了多个对象(一个数组), 它返回一个指向这些对象中的第一个对象的指针。如果对象的类型是 X, 由 new 返回的指针类型是 X*。例如:

```
char* q = new double[4]; // error: double* assigned to char*
```

new 返回一个指向一个 double 的指针, 而一个 double 并不是一个 char, 因此我们不能将它分配给指向 char 的变量 q。

17.4.1 自由空间分配

我们要求使用 new 操作符从空闲空间中分配内存:

- new 操作符返回一个指向被分配内存的指针。
- 一个指针的值是内存中首字节的地址。
- 一个指针指向一个特定类型的对象。
- 一个指针并不知道它指向多少个元素。

new 操作符可以为单个元素或一系列(数组)元素分配内存。例如:

```
int* pi = new int;           // allocate one int
int* qi = new int[4];        // allocate 4 ints (an array of 4 ints)

double* pd = new double;     // allocate one double
double* qd = new double[n];  // allocate n doubles (an array of n doubles)
```

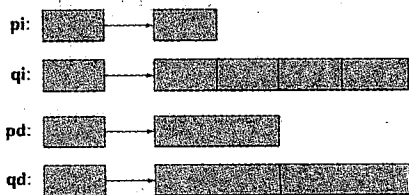
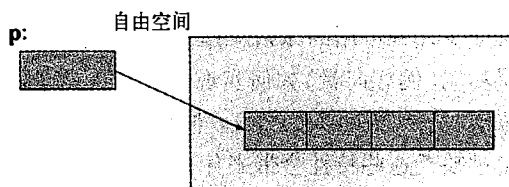
注意, 分配的对象数量可能是变化的。由于允许我们在运行时选择分配多少个对象, 因此它是很重要的。如果 n 等于 2, 我们得到右图。指向不同类型变量的指针类型不同。例如:

```
pi = pd; // error: can't assign a double* to an int*
pd = pi; // error: can't assign an int* to a double*
```

为什么不? 毕竟, 我们可以将 int 赋给 double 和将 double 赋给 int。原因在于[]操作符, 它依赖于元素

类型的大小, 以指出到哪里找到一个元素。例如, qi[2] 在内存中比 qi[0] 大两个 int 的大小, qd[2] 在内存中比 qd[0] 大两个 double 的大小。如果一个 int 与一个 double 的大小不同(在很多计算机中都是这样), 那么如果我们允许将 qi 指向分配给 qd 的内存, 我们将会得到一些奇怪的结果。

这是“实际的解释”。理论上的解释是“允许为指针分配不同类型将会导致类型错误”。



17.4.2 通过指针访问数据

另外，在一个指针上使用解引用操作符`*`，我们就可以使用下标操作符`[]`。例如：

```
double* p = new double[4];    // allocate 4 doubles on the free store
double x = *p;                // read the (first) object pointed to by p
double y = p[2];              // read the 3rd object pointed to by p
```

不出所料，下标操作符和向量的下标操作符一样都是从 0 开始计数，因此 `p[2]` 是第三个元素；`p[0]` 是第一个元素，因此 `p[0]` 实际上与 `*p` 相同。`[]` 和 `*` 操作符也可以用于写入：

```
*p = 7.7;                    // write to the (first) object pointed to by p
p[2] = 9.9;                  // write to the 3rd object pointed to by p
```

一个指针指向内存中的一个对象。“内容”操作符（又称为解引用操作符）允许我们读取或写入指针 `p` 指向的对象：

```
double x = *p;               // read the object pointed to by p
*p = 8.8;                    // write to the object pointed to by p
```

当我们使用一个指针时，`[]` 操作符将内存看做一系列的对象（其类型在指针声明时指定），并且用指针 `p` 指向第一个对象：

```
double x = p[3];             // read the 4th object pointed to by p
p[3] = 4.4;                  // write to the 4th object pointed to by p
double y = p[0];             // p[0] is the same as *p
```

这就是全部。这里没有检查和巧妙的实现，只是简单地访问我们计算机的内存：

p[0]:	p[1]:	p[2]:	p[3]:
8.8		9.9	4.4

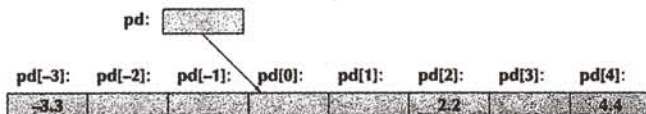
这确实是简单和最有效率的访问内存的机制，它是在实现一个向量时所需要的。

17.4.3 指针范围

指针带来的主要问题是一个指针并不“知道”它指向多少个元素。考虑下面的代码：

```
double* pd = new double[3];
pd[2] = 2.2;
pd[4] = 4.4;
pd[-3] = -3.3;
```

`pd` 是否有第三个元素 `pd[2]`？它是否有第五个元素 `pd[4]`？如果查看 `pd` 的定义，我们发现答案可以分别是“是”和“否”。但是，编译器不知道这些；它并不跟踪指针的值。如果我们分配足够多的内存，代码将会简单地访问内存。如果将三个 `double` 放在 `pd` 指向的内存部分之前，程序甚至会访问 `pd[-3]`：



我们并不知道 `pd[-3]` 与 `pd[4]` 被分配的内存位置。但是，即使我们知道也不意味着它们可以用于作为包含三个 `double` 指针的指针数组 `pd` 的一部分。最有可能的是，它们与其他对象的一部分，并且我们只是胡乱写入它们，这不是一个好主意。实际上，这是一个典型的灾难性的坏主意：“灾难性”表现在“程序神秘地崩溃”或“程序得到错误的输出”。尝试着大声说出来，尽管它听起来根本不好，我们要走很长一段路去避免它。越界访问是特别令人讨厌的，这是由于程序中无关的部分显然会受到影响。一次越界的读取会给我们一个“随机”值，这可能依赖于某些完全无关的计

算。一次越界的写入会将某些对象变成“不可能”的状态，或者简单地为其赋予一个不期望和错误的值。这种写入直到发生后很长时间通常不会被注意到，因此它们很难被发现。更糟糕的是：你运行一个带有越界错误的程序两次，输入稍有不同就可能出现不同的结果。这种错误（“临时性错误”）是最难发现的错误之一。

我们必须保证不出现这种越界的访问。我们使用 `vector` 而不是直接使用 `new` 来分配内存的原因之一是 `vector` 知道它的大小，这样就很容易避免越界的访问。

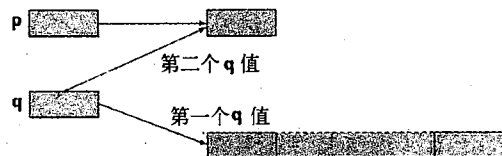
避免越界的访问变得困难的是，我们可以将一个 `double*` 赋予另一个 `double*`，而不必去管每个指针指向多少个对象。一个指针实际上并不知道它指向多少个对象。例如：

```
double* p = new double;           // allocate a double
double* q = new double[1000];     // allocate 1000 doubles

q[700] = 7.7;                     // fine
q = p;                             // let q point to the same as p
double d = q[700];                 // out-of-range access!
```

这里只有 3 行代码，`q[700]` 涉及两个不同的内存位置，最后一次使用是越界的访问，并且是一个

可能出现的灾难，如右图所示。现在，我们希望你会问“为什么指针不会记住自己的大小”？很显然，我们可以设计一个可以记住大小的“指针”，向量就是这样的指针。如果你阅读 C++ 文献和库，你将会发现很多“聪明的指针”可以弥补这些



低层次指针的缺陷。但是，有时我们需要接触硬件层次和了解对象如何编址，一台机器的地址并不知道它指向的是什么。另外，理解指针与理解大量的实际代码同样重要。

17.4.4 初始化

我们永远要保证使用对象之前为它赋一个值，也就是说，我们希望确认指针被初始化，并且指针指向的对象被初始化。思考下面的代码：

```
double* p0;                       // uninitialized: likely trouble
double* p1 = new double;          // get (allocate) an uninitialized double
double* p2 = new double(5.5);     // get a double initialized to 5.5
double* p3 = new double[5];       // get (allocate) 5 uninitialized doubles
```

很明显，声明 `p0` 但没有初始化它会带来麻烦。如下所示：

```
*p0 = 7.0;
```

本行代码将 7.0 赋给内存中的某些位置。我们并不知道将会是哪部分内存。它可能是无害的，但是永远不要这样做。我们迟早会得到与越界的访问相同的结果：“程序神秘地崩溃”或“程序得到错误的输出”。对于老式 C++ 程序（“C 风格程序”），严重错误中的很大比例是由未初始化指针的访问或越界的访问而引起的。我们必须尽最大努力去避免这种访问，一部分原因是我们着眼于专业化，另一部分原因是我们不想浪费时间查找这种错误。很少有事情像查找这种错误一样令人沮丧和厌倦。避免错误而不是查找它更令人愉快和有效率。

使用 `new` 分配内存不会初始化内置的类型。如果你不想对单独的对象这样做，你可以像对 `p2` 所做的那样指定值：`*p2` 变成了 5.5。注意使用 `()` 来初始化，这对照于使用 `[]` 来标识“数组”。

为由 `new` 分配的内置类型的数组对象指定初始化并不方便。对于数组来说，如果我们不喜欢默认的初始化，我们就必须自己做一些工作。例如：

```
double* p4 = new double[5];
for (int i = 0; i < 5; ++i) p4[i] = i;
```


现在, `p4` 指向 `double` 类型的变量, 包括 0.0、1.0、2.0、3.0 和 4.0。

像往常一样, 我们应该关心未初始化的对象, 并确认我们在读取它之前已为它赋值。注意编译器经常有一个“调试模式”, 每个变量被默认初始化成一个预期值(通常为 0)。这意味着当关闭调试模式并运行程序时, 当运行一个优化的程序时, 或只是简单地不同机器上编译时, 带有未初始化变量的程序可能马上显示出差异。不要被未初始化的变量牵着鼻子走。

当我们定义自己的类型时, 我们可以更好地控制初始化。如果一个类型 `X` 有一个默认的构造函数, 我们得到:

```
X* px1 = new X;           // one default-initialized X
X* px2 = new X[17];       // 17 default-initialized Xs
```

如果一个类型 `Y` 有一个构造函数, 但不是默认的构造函数, 我们需要显式地初始化:

```
Y* py1 = new Y;           // error: no default constructor
Y* py2 = new Y[17];       // error: no default constructor
Y* py3 = new Y(13);       // OK: initialized to Y(13)
```

17.4.5 空指针

如果你没有其他指针用于初始化一个指针, 那么使用 0:

```
double* p0 = 0;           // the null pointer
```

当 0 被赋给一个指针时, 0 被称为空指针。我们经常通过检测指针是否为 0, 以判断一个指针是否有效(如它是否指向什么东西)。例如:

```
if (p0 != 0) // consider p0 valid
```

这并不是一个完美的测试, 这是由于 `p0` 可能包含一个碰巧不是 0 的“随机”值, 或者一个已经被删除对象的地址(参见 17.4.6 节)。但是, 它经常是我们所能做得最好的。我们实际上不会明确地提到 0, 这是由于 `if` 语句会检测是否它的条件为非 0:

```
if (p0) // consider p0 valid; equivalent to p0!=0
```

考虑到它更直接表达“`p0` 有效”的思想, 我们比较喜欢这种短的模式, 但是也有不同意见。

当我们有一个指针有时指向一个对象而有时不指向时, 我们就需要使用空指针。它比很多人的想法更少见, 思考一下: 如果你并没有一个对象由指针来引用, 为什么你需要定义那个指针? 你不能等到有一个对象时再做吗?

17.4.6 自由空间释放

`new` 操作符会从自由空间中分配内存。由于一台计算机的内存是有限的, 因此在使用结束后将内存释放回自由空间通常是一个好主意。这样自由空间可以将这些内存重新用于新的分配。对于大型的程序和长时间运行的程序来说, 这种自由空间的重新使用是很重要的。例如:

```
double* calc(int res_size, int max) // leaks memory
{
    double* p = new double[max];
    double* res = new double[res_size];
    // use p to calculate results to be put in res
    return res;
}

double* r = calc(100, 1000);
```

在写操作时, 每次调用 `calc()` 会造成分配给 `p` 的 `double` 数组“泄漏”。例如, 调用 `calc(100, 1000)` 将会分配 100 个 `double` 的空间, 而这些空间无法被程序其他部分使用。

将内存返回自由空间的操作符称为 `delete`。我们对一个指针使用 `delete` 返回用 `new` 分配的内存, 以使这些内存可以用于未来的分配。现在, 这个例子变为:

```
double* calc(int res_size, int max)
// the caller is responsible for the memory allocated for res
{
    double* p = new double[max];
    double* res = new double[res_size];
    // use p to calculate results to be put in res
    delete[] p;    // we don't need that memory anymore: free it
    return res;
}

double* r = calc(100,1000);
// use r
delete[] r;    // we don't need that memory anymore: free it
```

顺便说一句，这个例子证明使用自由内存的一个主要原因：我们可以在一个函数中创建对象，并将它们传送给函数的调用者。

这里有两种形式的 delete：

- delete p 释放由 new 分配给单个对象的内存。
- delete[] p 释放由 new 分配给数组对象的内存。

对于程序员来说，使用正确的方式是一件乏味的工作。

删除一个对象两次是一个糟糕的错误。例如：

```
int* p = new int(5);
delete p;    // fine: p points to an object created by new
// ... no use of p here ...
delete p;    // error: p points to memory owned by the free-store manager
```

第二个 delete p 带来两个问题：

- 因此自由空间管理器可能会改变它的内部数据结构，导致无法再次正确地执行 delete p，你已不再拥有指针所指向的对象。
- 自由空间管理可能已“回收”p 指向的内存，因此 p 现在可能指向其他对象；删除其他对象（由程序的其他部分所拥有）将会在你的程序中引起错误。

这两个问题都发生在实际的程序中；它们不只是在理论上有可能性。

删除空指针不会做任何事（因为空指针不指向一个对象），因此删除空指针是无害的。例如：

```
int* p = 0;
delete p;    // fine: no action needed
delete p;    // also fine (still no action needed)
```

为什么我们都会被自由内存所困扰？编译器不能指出我们什么时候不需要一段内存，并在没有人工干预的情况下将它回收吗？它可以。这个过程称为自动垃圾收集或垃圾收集。不幸的是，自动垃圾收集并不是免费的，并且不是对所有应用都是理想的。如果你实际需要自动垃圾收集，你可以将一个自动垃圾收集器插入你的 C++ 程序。好的垃圾收集器是有效的（见 www.research.att.com/~bs/C++.html）。但是，本书假设你需要自己处理“垃圾”，并且我们教你如何方便和有效地来做。

在什么时候不泄漏内存是重要的？一个“永远”运行的程序不能承受泄漏内存。一个不能有内存泄漏的操作系统是“永远运行的”程序的例子，大多数的嵌入式系统（参见第 25 章）也是这样。很多程序使用库作为系统的一部分，因此库也不能出现内存泄漏的问题。一般来说，所有程序都不产生内存泄漏是个好主意。很多程序员将泄漏的原因归结于马虎，但是，这并没有切中要点。当你在一种操作系统（UNIX、Windows 等）上运行程序，在程序结束时会将所有内存自动返回给系统。这会带来一个问题，如果你知道你的程序不会使用比可用更多的内存，你可能“合理地”

决定泄漏内存直到操作系统为你释放内存。但是，如果你决定要这样做，确定你所估计的内存消耗是正确的，否则人们将会有好的理由认为你是草率的。

17.5 析构函数

现在，我们知道如何为一个向量保存元素。我们简单地对这些元素在自由空间中分配足够的空间，并且通过一个指针来访问它们：

```
// a very simplified vector of doubles
class vector {
    int sz;           // the size
    double* elem;     // a pointer to the elements
public:
    vector(int s)      // constructor
        :sz(s),        // initialize sz
        elem(new double[s]) // initialize elem
    {
        for (int i=0; i<s; ++i) elem[i]=0; // initialize elements
    }
    int size() const { return sz; } // the current size
    // ...
};
```

因此，sz 是元素的数量。我们在构造函数中初始化它，用户可以通过调用 size() 得到向量中的元素数量。在构造函数中使用 new 来分配元素空间，从自由空间返回的指针被保存在成员指针 elem 中。

注意，我们将这些元素初始化为它们的默认值(0.0)。标准库向量会这样做，因此我们认为最好从开始就这样做。

不幸的是，我们最初的 vector 会泄漏内存。在构造函数中，它使用 new 来为元素分配内存。遵循在 17.4 节中描述的规则，我们必须确认使用 delete 释放这些内存。思考下面的代码：

```
void f(int n)
{
    vector v(n); // allocate n doubles
    // ...
}
```

当我们离开函数 f() 时，v 在自由空间中创建的元素没有释放。我们可以为 vector 定义一个 clean_up() 操作并调用它：

```
void f2(int n)
{
    vector v(n); // define a vector (which allocates another n ints)
    // ... use v ...
    v.clean_up(); // clean_up() deletes elem
}
```

它可以工作。但是，有关自由空间的一个最常见的问题是人们忘记 delete。clean_up() 可能会带来相关的问题；人们可能忘记调用它。我们可以做得更好，基本思路是使编译器知道一个函数可以做与构造函数相反的功能，就像编译器了解构造函数一样。这个函数不可避免地被称作析构函数。同样，在一个对象的类创建时会隐式调用构造函数，当一个对象离开作用域时会隐式调用析构函数。构造函数用于确认一个对象是否被正确创建和初始化。与之相反，析构函数用于确认一个对象是否被正确销毁。例如：

```
// a very simplified vector of doubles
class vector {
    int sz;           // the size
    double* elem;     // a pointer to the elements
public:
    vector(int s)      // constructor
        :sz(s), elem(new double[s]) // allocate memory
    {
        for (int i=0; i<s; ++i) elem[i]=0; // initialize elements
    }

    ~vector()          // destructor
    { delete[] elem; } // free memory

    // ...
};
```

有了这个定义，我们就可以这样使用 vector 了：

```
void f3(int n)
{
    int* p = new int[n]; // allocate n ints
    vector v(n);         // define a vector (which allocates another n ints)
    // ... use p and v ...
    delete[] p;          // deallocate the ints
} // vector automatically cleans up after v
```

delete[] 看起来相当繁琐并且容易出错。对于 vector，我们不必使用 new 分配内存，以及在函数结束时使用 delete[] 释放内存。vector 已经做了这些工作，而且做得更好。特别是，vector 不能忘记调用它的析构函数释放元素使用的内存。

我们在这里并不打算介绍析构函数使用的更多细节，但是它们对于处理资源来说很重要，这些资源需要使用前申请和使用后释放：文件、线程、锁等。记住 iostream 如何在使用后清除自己，它们刷新缓冲区、关闭文件、释放缓冲区空间等。这些工作由它们的析构函数完成，每个“拥有”资源的类都需要一个析构函数。

17.5.1 生成的析构函数

如果一个类的成员拥有一个析构函数，则在包含这个成员的对象销毁时调用这个析构函数。例如：

```
struct Customer {
    string name;
    vector<string> addresses;
    // ...
};

void some_fct()
{
    Customer fred;
    // initialize fred
    // use fred
}
```

当我们退出函数 some_fct() 时，fred 将会离开作用域，因此 fred 要被销毁；也就是说，name 和 addresses 的析构函数被调用。这个过程对于析构函数的可用性明显很重要，它有时被表达为“编译器为 Customer 生成一个析构函数，它调用成员的析构函数”。“析构函数应被调用”这一显然而必要的保证通常就是如此实现的。

成员和基类的析构函数在从派生类的析构函数（无论用户定义或编译器生成）中被隐式调用。基本上，所有的规则可以被总结为：“当对象被销毁时，析构函数被调用”（当离开作用域时、当

delete 被调用时等)。

17.5.2 析构函数和自由空间

析构函数从概念上来说简单的,但它是大多数有效的 C++ 编程技术的基础。它的基本思路是简单的:

- 无论一个类对象需要使用哪种资源,这种资源都要在构造函数中获得。
- 在对象的生命期中,它可以释放资源和获得新的资源。
- 在对象的生命期结束后,析构函数释放对象拥有的所有资源。

在 vector 中处理自由空间的一对构造函数、析构函数是一个典型的例子。我们将在 19.5 节中提供这种思路的更多例子。在这里,我们将测试一个重要的应用,它是自由空间使用和类层次关系的结合。思考下面的代码:

```
Shape* fct()
{
    Text tt(Point(200,200),"Annemarie");
    // ...
    Shape* p = new Text(Point(100,100),"Nicholas");
    return p;
}

void f()
{
    Shape* q = fct();
    // ...
    delete q;
}
```

这看起来相当合理,确实是这样。它完全工作正常,让我们通过分析它是如何工作的来揭示一些精湛、重要和简单的技术。在函数 fct() 中,Text 对象 tt 在离开 fct() 时被销毁。Text 有一个 string 成员,很明显需要调用它的析构函数, string 像 vector 一样处理内存的获取和释放。因此对于 tt 来说是容易的;编译器只需像 17.5.1 节中描述的那样调用 Text 生成的析构函数。但是,从 fct() 返回的 Text 对象是什么?调用函数 f() 完全不知道 q 指向一个 Text;它只知道 q 指向一个 Shape。delete q 如何调用 Text 的析构函数?

在 14.2.1 节中,我们快速掠过一事实:Shape 有一个析构函数。实际上,Shape 有一个虚的析构函数,这正是问题的关键。当我们使用 delete q 时,delete 会查看 q 的类型,以确定是否需要调用析构函数,如果是就调用它。因此,delete q 调用 Shape 的析构函数 ~Shape()。但是,~Shape() 是虚函数,因此使用虚函数调用机制(见 14.3.1 节),这个调用会引用 Shape 的派生类的析构函数,在这里是 ~Text()。如果 Shape::~Shape() 不是虚函数,Text::~Text() 将不会被调用,并且 Text 的 string 成员将不会被销毁。

作为一个经验法则:如果你有一个带有虚函数功能的类,则它需要一个虚的析构函数。具体原因是:

- 1) 如果一个类有虚函数功能,它经常作为一个基类使用。
- 2) 如果它是一个基类,它的派生类经常使用 new 来分配。
- 3) 如果一个派生类对象使用 new 来分配,并且通过一个指向它的基类的指针来控制,那么它经常通过一个指向它的基类的指针来删除它。

注意,析构函数是通过 delete 来隐式或间接调用。它们并不是直接调用的。这样会省去很多麻烦的工作。

试一试 编写一个使用基类和成员的小程序，自己定义构造函数和析构函数，在它们被调用时输出一行信息。然后，创建几个对象并查看它们的构造函数和析构函数如何被调用。

17.6 访问向量元素

为了使 `vector` 可以使用，我们需要一种读和写元素的方法。对于初学者来说，我们可以提供简单的 `get()` 和 `set()` 成员函数：

```
// a very simplified vector of doubles
class vector {
    int sz;           // the size
    double* elem;     // a pointer to the elements
public:
    vector(int s) :sz(s), elem(new double[s]) { } // constructor
    ~vector() { delete[] elem; } // destructor
    int size() const { return sz; } // the current size

    double get(int n) { return elem[n]; } // access: read
    void set(int n, double v) { elem[n]=v; } // access: write
};
```

`get()` 和 `set()` 都可以访问元素，在 `elem` 指针上使用 `[]` 操作符：`elem[n]`。

现在，我们可以生成一个 `double` 型的 `vector` 并使用它：

```
vector v(5);
for (int i=0; i<v.size(); ++i) {
    v.set(i, 1.1*i);
    cout << "v[" << i << "]==" << v.get(i) << '\n';
}
```

这里将会输出：

```
v[0]==0
v[1]==1.1
v[2]==2.2
v[3]==3.3
v[4]==4.4
```

这仍是一个相当简单的 `vector`，相对于常用的下标符号来说，使用 `get()` 和 `set()` 的代码是很难看的。但是，我们的目的是从小的和简单的程序开始，沿着这个方式逐步测试和扩充我们的程序。这种发展和反复测试的策略可以减少错误和调试过程。

17.7 指向类对象的指针

“指针”的概念是通用的，因此我们可以指向可分配内存的任何东西。例如，我们可以使用指向 `vector` 的指针，就像我们使用指向 `char` 的指针：

```
vector* f(int s)
{
    vector* p = new vector(s); // allocate a vector on free store
    // fill *p
    return p;
}

void ff()
{
    vector* q = f(4);
    // use *q
    delete q; // free vector on free store
}
```

注意, 当我们删除一个 `vector` 时, 它的析构函数会被调用。例如:

```
vector* p = new vector(s);    // allocate a vector on free store
delete p;                    // deallocate
```

在自由空间中创建 `vector`, `new` 操作符:

- 首先为一个 `vector` 分配内存。
- 然后, 激活 `vector` 的构造函数初始化 `vector`; 构造函数为 `vector` 的元素分配内存, 并初始化这些元素。

删除 `vector`, `delete` 操作符:

- 首先激活 `vector` 的析构函数; 这个析构函数激活这些元素的析构函数(如果它们有析构函数), 然后释放这些元素使用的内存。
- 然后, 释放 `vector` 使用的内存。

注意, 这个工作可以完美地递归执行(见 8.5.8 节)。我们也可以使用实际的(标准库)`vector` 来做到:

```
vector<vector<double>>* p = new vector<vector<double>>(10);
delete p;
```

这里, `delete p` 激活 `vector<vector<double>>` 的析构函数; 接着, 这个析构函数激活它的 `vector<double>` 元素的析构函数, 所有相关的东西完全被清除, 不会留下未销毁的对象和泄漏的内存。

由于 `delete` 激活析构函数(为某种类型, 例如 `vector`), `delete` 通常称为销毁对象, 而不只是释放它们。

同样, 请记住一个位于函数之外的“单独的”`new`, 将会带来忘记删除它的机会。除非你有一个删除对象的好(确实简单, 例如 13.10 节和附录 E.4 中的 `Vector_ref`)的策略, 尽量将 `new` 放在构造函数中, 以及将 `delete` 放在析构函数中。

到目前为止一切很好, 但是, 如果仅有一个指针, 我们如何访问一个 `vector` 的成员? 注意, 给一个对象名, 所有类都支持通过 `.` (点)操作符来访问成员:

```
vector v(4);
int x = v.size();
double d = v.get(3);
```

与此类似, 给定一个对象指针, 所有类支持通过 `->` (箭头)操作符来访问成员:

```
vector* p = new vector(4);
int x = p->size();
double d = p->get(3);
```

`.` (点)和 `->` (箭头)可以用于数据成员和函数成员。由于内置类型(例如 `int` 和 `double`)没有成员, 因此 `->` 不能用于内置类型。点和箭头通常称为成员访问操作符。

17.8 类型混用: 无类型指针和指针类型转换

在使用指针和自由空间分配的数组时, 我们非常接近硬件层面。基本上, 我们对指针的操作(初始化、分配、`*` 和 `[]`)直接映射为机器指令。在这个层次, 语言只提供一点描述上的便利, 以及由类型系统提供的编译时的一致性。偶尔, 我们不得不放弃这最后一点保护。

通常, 我们不希望在没有类型系统的保护下工作, 但是有时没有其他选择(例如, 我们需要与无法识别 C++ 类型的其他语言交互)。我们有时也会遇到一些情况, 我们需要面对没有根据安全类型设计的老的代码。在这种情况下, 我们需要两样东西:

- 一种指向不知道自己保存的是何种对象的内存的指针。
- 一种操作, 对于这类指针, 它告诉编译器指针指向的是哪种(未证实)的操作类型。

类型 `void*` 的含义是“指向编译器不知道类型的那些内存”。当我们在两段代码之间传输一个地址，并且不知道每段代码实际的类型时，就可以使用 `void*`。这方面的例子包括在回调函数（见 16.3.1 节）讨论的“地址”和最低层的内存分配器（例如 `new` 操作符的实现）。

不存在 `void` 类型的对象，但是正如我们看到的，通常用 `void` 来表示“没有返回值”：

```
void v;           // error: there are no objects of type void
void f();         // f() returns nothing — f() does not return an object of type void
```

我们可以将指向任意对象的指针赋予 `void*`。例如：

```
void* pv1 = new int;           // OK: int* converts to void*
void* pv2 = new double[10];    // OK: double* converts to void*
```

由于编译器不知道 `void*` 指向什么，因此我们必须告诉它：

```
void f(void* pv)
{
    void* pv2 = pv;    // copying is OK (copying is what void*s are for)
    double* pd = pv;   // error: cannot convert void* to double*
    *pv = 7;           // error: cannot dereference a void*
                      // (we don't know what type of object it points to)
    pv[2] = 9;         // error: cannot subscript a void*
    int* pi = static_cast<int*>(pv); // OK: explicit conversion
    // ...
}
```

`static_cast` 可以用于两种相关指针类型之间的强制转换，例如 `void*` 与 `double*`（见附录 A.5.7）。“`static_cast`”确实是一个丑陋的名字，表示一个丑陋的操作，我们只在完全有必要时才使用它，通常你会发现它没有必要。一个操作（例如 `static_cast`）称为明确的类型转换（因为这就是它所做的事）或口语化地称为类型转换（因为它常用于被破坏的数据）。

C++ 提供两个潜在的比 `static_cast` 更危险的转换：

- `reinterpret_cast` 可以在两个不相关的类型之间转换，例如 `int` 和 `double*`。
- `const_cast` 可以“抛弃常量修饰”。

例如：

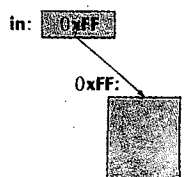
```
Register* in = reinterpret_cast<Register*>(0xff);

void f(const Buffer* p)
{
    Buffer* b = const_cast<Buffer*>(p);
    // ...
}
```

第一个例子是有关 `reinterpret_cast` 使用的经典例子。我们告诉编译器，内存中的某个特定部分（开始于 `0xFF` 位置的内存，如右图所示。）告诉编译器，这部分内存被用作一个寄存器（可能结合特定的语法使用）。在我们写类似于设备驱动程序时，采用这种代码是必要的。

在第二个例子中，`const_cast` 将名为 `p` 的 `const Buffer*` 的属性抛弃。我们大概知道自己在做什么。

`static_cast` 至少不会混淆指针/整数或出现“消除 `const`”，因此在你感到有必要使用类型转换时最好用 `static_cast`。当你认为需要进行转换时，重新考虑：是否有办法书写代码而不使用转换？是否有办法重新设计程序部分而不使用转换？除非你需要与其他人的代码或硬件打交道，通常都会有办法避免类型转换。否则，你将会面对微妙和讨厌



的错误。你不要期望使用 `reinterpret_cast` 的代码可以移植。

17.9 指针和引用

我们可以将一个引用看做是一个自动解引用的、不可改变的指针或是一个对象的别名。指针和引用在以下几个方面不同：

- 为一个指针赋值会改变指针的值(不是指针指向的值)。
- 为了得到一个指针,你通常需要使用 `new` 或 `&`。
- 为了访问一个指针指向的对象,你可以使用 `*` 或 `[]`。
- 为一个引用赋值会改变引用指向的值(不是引用自身的值)。
- 在初始化一个引用之后,你不能让引用指向其他对象。
- 为引用赋值执行深度复制(赋值给引用的对象);为指针赋值不是这样(赋值给指针自身)。
- 注意空指针。

例如:

```
int x = 10;
int* p = &x;    // you need & to get a pointer
*p = 7;         // use * to assign to x through p
int x2 = *p;     // read x through p
int* p2 = &x2;   // get a pointer to another int
p2 = p;         // p2 and p both point to x
p = &x2;        // make p point to another object
```

引用的相关例子如下:

```
int y = 10;
int& r = y;     // the & is in the type, not in the initializer
r = 7;         // assign to y through r (no * needed)
int y2 = r;     // read y through r (no * needed)
int& r2 = y2;   // get a reference to another int
r2 = r;        // the value of y is assigned to y2
r = &y2;       // error: you can't change the value of a reference
               // (no assignment of an int* to an int&)
```

注意最后一个例子,它是不正确的,这个语法结构将会失败。这是由于初始化后无法让引用指向不同的对象,如果你需要指向不同的对象,请使用指针。了解如何使用指针,参见 17.9.3 节。

引用和指针都是通过使用内存地址来实现的。它们只是在地址的使用上不同,为编程人员提供稍有不同的功能。

17.9.1 指针参数和引用参数

当你希望将一个变量的值改为由函数计算出的结果时,你有 3 种选择。例如:

```
int incr_v(int x) { return x+1; } // compute a new value and return it
void incr_p(int* p) { ++*p; }    // pass a pointer
                                   // (dereference it and increment the result)
void incr_r(int& r) { ++r; }     // pass a reference
```

你会如何选择呢?我们认为返回值是最明显的代码(因此最不容易出错),那就是:

```
int x = 2;
x = incr_v(x); // copy x to incr_v(); then copy the result out and assign it
```

对于小的对象,例如一个 `int`,我们倾向使用这种风格。但是,返回或向前传一个值并不总是可行。例如,我们可以编写一个函数来修改一个大的数据结构,例如一个包含 10 000 个 `int` 的向量;我们不能以可接受的效率拷贝 40 000 字节(至少两次)。

我们如何选择使用引用参数还是指针参数呢？不幸的是，每种方法都有自己的优点和缺点，因此在这方面仍没有明确的答案。你需要根据程序功能和可能的用途来决定。

使用指针可以提醒编程者有些东西可能改变。例如：

```
int x = 7;
incr_p(&x)  // the & is needed
incr_r(x);
```

在 `incr_p(&x)` 中使用 `&` 通知用户 `x` 可能改变。与之对比，`incr_r(x)` “看起来很无辜”。这导致很多人稍微偏爱使用指针。

另一方面，如果你使用指针作为函数的参数，需要防止某些人使用空指针来调用函数，空指针是指值为零的指针。例如：

```
incr_p(0);  // crash: incr_p() will try to dereference 0
int* p = 0;
incr_p(p);  // crash: incr_p() will try to dereference 0
```

这明显是很讨厌的。编写 `incr_p()` 的人可以防止它：

```
void incr_p(int* p)
{
    if (p==0) error("null pointer argument to incr_p()");
    ++*p;  // dereference the pointer and increment the object pointed to
}
```

但是，`incr_p()` 突然不像以前看起来简单和有吸引力。第 5 章讨论如何处理这个不好的问题。与之相比，引用（例如 `incr_r()`）的用户有权假设引用指向一个有效的对象。

如果“不传递任何东西”（不传递对象）可以被函数接受，那么我们应该使用指针参数。注意，这并不是针对递增操作的情况，它需要为 `p == 0` 抛出一个异常。

因此，实际的答案是“选择依赖于函数的性质”：

- 对于小的对象，倾向于传递值。
- 对于“没有对象”（用 0 表示）是有效参数的函数，使用一个指针参数（记着对 0 进行测试）。
- 否则，使用一个引用参数。

参见 8.5.6 节。

17.9.2 指针、引用和继承

在 14.3 节中，我们看到一个派生类（例如 `Circle`）如何被用在需要它的公有基类 `Shape` 的对象的地方。我们可以用指针或引用来表达这个思想：由于 `Shape` 是 `Circle` 的一个公有基类，因此 `Circle*` 可以被强制转换为 `Shape*`。例如：

```
void rotate(Shape* s, int n);  // rotate *s n degrees

Shape* p = new Circle(Point(100,100),40);
Circle c(Point(200,200),50);
rotate(&c,45);
```

对于引用是类似的：

```
void rotate(Shape& s, int n);  // rotate s n degrees

Shape& r = c;
rotate(c,75);
```

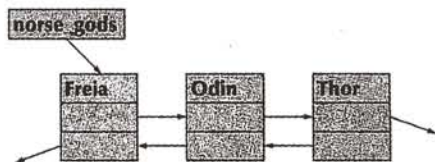
这是大多数的面向对象的编程技术的关键（参见 14.3 节和 14.4 节）。

17.9.3 实例：列表

列表是最普通和有用的数据结构之一。列表通常会有一些“链接”，每个链接会保存一些信息

和指向其他链接的指针,这是指针的典型用途之一。例如,一个短的列表 `norse_gods` 可表示如下:一个像这样的列表称为双向链表(doubly-linked list),我们可以在一个链接上找到前趋与后继。一个只能找到后继的列表称为单向链表(singly-linked list)。当我们希望很容易地移除一个元素时,我们可以使用双向链表来实现。我们可以定义链接如下:

```
struct Link {
    string value;
    Link* prev;
    Link* succ;
    Link(const string& v, Link* p = 0, Link* s = 0)
        : value(v), prev(p), succ(s) {}
};
```



这样,给定一个链接,我们可以使用 `succ` 指针到达它的后继,或者使用 `prev` 指针到达它的前趋。我们使用空指针来表示一个没有后继或前趋的链接。我们可以构造自己的列表 `norse_gods` 如下:

```
Link* norse_gods = new Link("Thor",0,0);
norse_gods = new Link("Odin", 0, norse_gods);
norse_gods->succ->prev = norse_gods;
norse_gods = new Link("Freia",0, norse_gods);
norse_gods->succ->prev = norse_gods;
```

我们通过创建多个链接来建立列表,并如图中所示将这些链接连起来:首先是 Thor,然后 Odin 是 Thor 的前趋,最后 Freia 是 Odin 的前趋。你可以跟随指针去查看我们已经正确建立了链表,这样每个后继和前趋都指向正确的位置。但是,这段代码是模糊的,这是由于我们没有显式定义和命名一个插入操作:

```
Link* insert(Link* p, Link* n)    // insert n before p (incomplete)
{
    n->succ = p;                // p comes after n
    p->prev->succ = n;           // n comes after what used to be p's predecessor
    n->prev = p->prev;           // p's predecessor becomes n's predecessor
    p->prev = n;                 // n becomes p's predecessor
    return n;
}
```

当 `p` 确实指向一个 Link,而 `p` 指向的 Link 确实有一个前趋时, `insert` 就可以正确工作。当我们思考指针与链接结构(例如由链接组成的列表)时,我们总是用小的带有框和箭头的图来验证代码对小的实例是否正确工作。请不要太骄傲,以至于不屑使用。

这个版本的 `insert()` 是不完整的,这是由于它没有处理 `n`、`p` 或 `p->prev` 为 0 的情况。我们增加适当的空指针测试,得到更繁杂但正确的版本:

```
Link* insert(Link* p, Link* n)    // insert n before p; return n
{
    if (n==0) return p;
    if (p==0) return n;
    n->succ = p;                // p comes after n
    if (p->prev) p->prev->succ = n;
    n->prev = p->prev;           // p's predecessor becomes n's predecessor
    p->prev = n;                 // n becomes p's predecessor
    return n;
}
```

有了这个 `insert` 函数,前文创建链表的代码就可以写成:

```
Link* norse_gods = new Link("Thor");
norse_gods = insert(norse_gods, new Link("Odin"));
norse_gods = insert(norse_gods, new Link("Freia"));
```

现在所有 `prev` 和 `succ` 指针易于引起的错误都从我们的视线中消失了。指针误用是乏味的和容易出错的,即使是编写良好并经过全面测试的函数中也会存在。特别是,常规代码中的很多错误是由于编程者忘记测试指针为 0 而引起的,正如我们(故意)在 `insert()` 的第一个版本中所做的那样。

注意,我们使用默认的参数(参见 15.3.1 节和附录 A.9.2),使用户不必每次使用构造函数时都要处理前趋与后继。

17.9.4 列表的操作

在标准库中提供了一个 `List` 类,我们将会在 20.4 节中介绍它。列表类中隐藏了所有的链接操作,但是这里我们将基于 `Link` 类详细说明列表的概念,以便查看在列表类的“表面背后”的内容,以及更多有关指针使用的例子。

我们的 `Link` 类需要哪些操作允许用户避免“指针错误”?这在某种程度上视开发者的偏好而定,但是这里有一组有用的操作:

- 构造函数。
- `insert`: 在一个元素前插入。
- `add`: 在一个元素后插入。
- `erase`: 删除一个元素。
- `find`: 查找一个给定值的 `Link`。
- `advance`: 获得第 n 个后继。

我们可以像这样编写这些操作:

```
Link* add(Link* p, Link* n)    // insert n after p; return n
{
    // much like insert (see exercise 11)
}

Link* erase(Link* p)           // remove *p from list; return p's successor
{
    if (p==0) return 0;
    if (p->succ) p->succ->prev = p->prev;
    if (p->prev) p->prev->succ = p->succ;
    return p->succ;
}

Link* find(Link* p, const string& s)    // find s in list;
                                         // return 0 for "not found"
{
    while(p) {
        if (p->value == s) return p;
        p = p->succ;
    }
    return 0;
}

Link* advance(Link* p, int n)           // move n positions in list
                                         // return 0 for "not found"
                                         // positive n moves forward, negative backward
{
    if (p==0) return 0;
    if (0<n) {
        while (n-->0) {
            if (p->succ == 0) return 0;
            p = p->succ;
        }
    }
    return p;
}
```

```

        p = p->succ;
    }
}
if (n<0) {
    while (n++) {
        if (p->prev == 0) return 0;
        p = p->prev;
    }
}
return p;
}

```

注意后缀 `n++` 的使用。这种递增(后递增)形式在自身值增加之前生成值。

17.9.5 列表的使用

作为一个小的练习，我们建立两个列表：

```

Link* norse_gods = new Link("Thor");
norse_gods = insert(norse_gods, new Link("Odin"));
norse_gods = insert(norse_gods, new Link("Zeus"));
norse_gods = insert(norse_gods, new Link("Freia"));

Link* greek_gods = new Link("Hera");
greek_gods = insert(greek_gods, new Link("Athena"));
greek_gods = insert(greek_gods, new Link("Mars"));
greek_gods = insert(greek_gods, new Link("Poseidon"));

```

不幸的是，我们犯了两个错误：Zeus 是一位希腊的天神，而不是一位北欧的天神；希腊的战争之神是 Ares，而不是 Mars (Mars 是他的拉丁/罗马名字)。我们可以修改它：

```

Link* p = find(greek_gods, "Mars");
if (p) p->value = "Ares";

```

注意，如何防止 `find()` 返回一个 0。我们认为在这种情况下它不可能发生(我们毕竟只是将 Mars 插入 `greek_gods` 中)，但在实际的例子中某些人可能改变代码。

与之相似，我们可以将 Zeus 移入正确的行列中：

```

Link* p = find(norse_gods, "Zeus");
if (p) {
    erase(p);
    insert(greek_gods, p);
}

```

你是否注意到了错误？它相当微妙(除非你直接使用过链接)。如果我们用 `erase()` 删除的链接是由 `norse_gods` 指向的，会发生什么？这种情况实际上不会发生，但是为了编写好的、可维护的代码，我们不得不考虑这种可能性。

```

Link* p = find(norse_gods, "Zeus");
if (p) {
    if (p==norse_gods) norse_gods = p->succ;
    erase(p);
    greek_gods = insert(greek_gods, p);
}

```

这里我们修改第二个错误：当我们在第一个希腊天神之前插入 Zeus 时，我们需要让 `greek_gods` 指向 Zeus 的链接。指针是非常有用和灵活的，但是也是相当微妙的。

最后，打印出这个列表：

```

void print_all(Link* p)
{
    cout << "{ ";
    while (p) {

```

```

        cout << p->value;
        if (p==p->succ) cout << ", ";
    }
    cout << " }";
}

print_all(norse_gods);
cout<<"\n";

print_all(greek_gods);
cout<<"\n";

```

将会得到:

```

{ Freia, Odin, Thor }
{ Zeus, Poseidon, Ares, Athena, Hera }

```

17.10 this 指针

注意, 在我们的每个列表函数中, 都使用一个 `Link*` 作为第一个参数, 并在这个对象中访问数据。我们经常使用的成员函数来实现这类函数。我们是否可以通过操作成员来简化 `Link` 呢? 我们是否可能使指针私有, 以便只有成员函数能够访问它呢? 我们可以的:

```

class Link {
public:
    string value;
    Link(const string& v, Link* p = 0, Link* s = 0)
        : value(v), prev(p), succ(s) {}

    Link* insert(Link* n);           // insert n before this object
    Link* add(Link* n);              // insert n after this object
    Link* erase();                   // remove this object from list
    Link* find(const string& s);     // find s in list
    const Link* find(const string& s) const; // find s in list
    Link* advance(int n) const;     // move n positions in list

    Link* next() const { return succ; }
    Link* previous() const { return prev; }

private:
    Link* prev;
    Link* succ;
};

```

这种方法看起来很有前途。将那些不能修改 `Link` 对象状态的操作定义为 `const` 成员函数。我们增加了两个(非修改性)函数 `next()` 和 `previous()`, 可供用户遍历列表(链表)。由于已经禁止用户直接访问 `succ` 和 `prev`, 这两个函数是必要的。数据值仍然定义为公有成员, 因为(到目前为止)我们没有理由不这么做: 它们“只是数据而已”。

现在我们复制之前的全局函数 `insert()`, 进行适当的修改来实现 `Link::insert()`:

```

Link* Link::insert(Link* n) // insert n before p; return n
{
    Link* p = this;         // pointer to this object
    if (n==0) return p;     // nothing to insert
    if (p==0) return n;     // nothing to insert into
    n->succ = p;             // p comes after n
    if (p->prev) p->prev->succ = n;
    n->prev = p->prev;       // p's predecessor becomes n's predecessor
    p->prev = n;             // n becomes p's predecessor
    return n;
}

```

但是,对于调用 `Link::insert()` 的那个对象,我们如何获得其指针呢?没有程序设计语言的帮助,我们是无法获得此指针的。不过,幸运的是,C++ 提供了相应的机制:在每个成员函数中,标识符 `this` 就是指向用户调用成员函数所用对象的指针。因此,可以简单地将代码中的 `p` 都替换为 `this`:

```
Link* Link::insert(Link* n)    // insert n before this object; return n
{
    if (n==0) return this;
    if (this==0) return n;
    n->succ = this;             // this object comes after n
    if (this->prev) this->prev->succ = n;
    n->prev = this->prev;        // this object's predecessor
                                // becomes n's predecessor
    this->prev = n;              // n becomes this object's predecessor
    return n;
}
```

这有些啰嗦,但 C++ 提供了更简单的语法,当我们访问当前对象的成员时,无需再写 `this`:

```
Link* Link::insert(Link* n)    // insert n before this object; return n
{
    if (n==0) return this;
    if (this==0) return n;
    n->succ = this;             // this object comes after n
    if (prev) prev->succ = n;
    n->prev = prev;             // this object's predecessor becomes n's predecessor
    prev = n;                  // n becomes this object's predecessor
    return n;
}
```

换句话说,只要是在访问成员,就可以省略指向当前对象的指针 `this`。只有当需要引用整个对象时,我们才需要显式使用 `this`。

注意 `this` 有特殊的意义:它指向用户调用成员函数时所用的对象。它不指向任何旧的对象。编译器保证我们在成员函数中不能修改 `this` 的值。例如:

```
struct S {
    // ...
    void mutate(S* p)
    {
        this = p; // error: "this" is immutable
        // ...
    }
};
```

17.10.1 关于 Link 使用的更多讨论

在处理实现的细节之后,我们可以看到使用链表的新代码如下:

```
Link* norse_gods = new Link("Thor");
norse_gods = norse_gods->insert(new Link("Odin"));
norse_gods = norse_gods->insert(new Link("Zeus"));
norse_gods = norse_gods->insert(new Link("Freia"));

Link* greek_gods = new Link("Hera");
greek_gods = greek_gods->insert(new Link("Athena"));
greek_gods = greek_gods->insert(new Link("Mars"));
greek_gods = greek_gods->insert(new Link("Poseidon"));
```

它与前面的版本非常相似。像前面的例子一样,我们改正所有的“错误”。修改战争之神的名字:

```
Link* p = greek_gods->find("Mars");
if (p) p->value = "Ares";
```

将 Zeus 移到正确的位置:

```
Link* p2 = norse_gods->find("Zeus");
if (p2) {
    if (p2==norse_gods) norse_gods = p2->next();
    p2->erase();
    greek_gods = greek_gods->insert(p2);
}
```

最后, 打印出这个列表:

```
void print_all(Link* p)
{
    cout << "{ ";
    while (p) {
        cout << p->value;
        if (p=p->next()) cout << ", ";
    }
    cout << " }";
}

print_all(norse_gods);
cout<<"\n";

print_all(greek_gods);
cout<<"\n";
```

将会得到:

```
{ Freia, Odin, Thor }
{ Zeus, Poseidon, Ares, Athena, Hera }
```

你更喜欢哪个版本: insert() 是成员函数的版本, 还是它是自由函数的版本? 在这种情况下, 差别并不大, 参见 9.7.5 节。

通过观察发现, 我们仍没有一个列表类, 只有一个链接类。这使得我们担心指向第一个元素的是哪个指针。我们可以通过定义一个 List 类来做得更好, 但是顺着这里给出的思路来设计是很简单的。在 20.4 节中将会介绍标准库 list。



简单练习

本章的简单练习包括两个部分。第一部分练习/建立对自由空间数组的理解, 并将数组与向量加以比较:

1. 使用 new 分配一个由 10 个 int 组成的数组。
2. 使用 cout 打印这 10 个 int 的值。
3. 使用 delete[] 释放这个数组。
4. 编写一个函数 print_array10(ostream& os, int* a), 将 a(假设包含 10 个元素) 的值打印到 os。
5. 分配一个由 10 个 int 组成的数组; 用值 100、101、102 等初始化数组; 打印数组的值。
6. 分配一个由 11 个 int 组成的数组; 用值 100、101、102 等初始化数组; 打印数组的值。
7. 编写一个函数 print_array(ostream& os, int* a, int n), 将 a(假设包含 n 个元素) 的值打印到 os。
8. 分配一个由 20 个 int 组成的数组; 用值 100、101、102 等初始化数组; 打印数组的值。
9. 你是否记得删除这个数组? (如果没有, 删除它。)
10. 重复做第 5、6、8 题, 使用一个向量来代替数组, 使用一个 print_vector() 来代替 print_array()。

第二部分集中在指针和它与数组的关系上。使用来自最后一个练习的 print_array():

1. 分配一个 int, 将它初始化为 7, 并将它的地址分配给变量 p1。
2. 打印 p1 的值和它指向的 int 的值。
3. 分配一个由 7 个 int 组成的数组; 将它初始化为 1、2、4、8 等; 将它的地址分配给变量 p2。
4. 打印 p2 的值和它指向的数组的值。

5. 声明一个名字为 p3 的 int*, 并使用 p2 来初始化它。
6. 将 p1 赋值给 p2。
7. 将 p3 赋值给 p2。
8. 打印 p1 和 p2 的值和它们指向的数组的值。
9. 释放所有通过自由空间分配的内存。
10. 分配一个由 10 个 int 组成的数组; 将它初始化为 1、2、4、8 等; 将它的地址分配给变量 p1。
11. 分配一个由 10 个 int 组成的数组, 并将它的地址赋值给变量 p2。
12. 将由 p1 指向的数组的值复制到由 p2 指向的数组。
13. 重做第 10~12 题, 使用一个向量来代替数组。



思考题

1. 为什么我们需要元素数量可变的数据结构?
2. 我们进行典型的编程时, 需要使用哪 4 种存储形式?
3. 什么是自由存储? 它常用的其他名称是什么? 哪种操作符支持它?
4. 什么是释放操作符, 为什么我们需要它?
5. 什么是地址? 在 C++ 中如何控制内存地址?
6. 指向一个对象的指针中包含什么信息? 它缺少什么有用的信息?
7. 一个指针可以指向什么?
8. 什么是内存泄漏?
9. 什么是资源?
10. 我们如何初始化一个指针?
11. 什么是空指针? 我们什么时候需要使用它?
12. 我们什么时候需要指针(而不是一个引用或命名对象)?
13. 什么是析构函数? 我们什么时候需要使用它?
14. 我们什么时候需要虚析构函数?
15. 成员如何来调用析构函数?
16. 什么是转换? 我们什么时候需要使用它?
17. 什么是双向的列表?
18. 什么是 this? 我们什么时候需要使用它?



术语

地址	析构函数	指针	地址: &	自由空间	范围
分配	链接	资源泄漏	转换	列表	子脚本
容器	成员访问: ->	子脚本: []	内容: *	成员析构函数	this
释放	内存	类型转换	delete	内存泄漏	虚析构函数
delete[]	new	void*	解引用	空指针	



习题

1. 什么是你的实现中的指针值的输出形式? 提示: 不要阅读相关文档。
2. 一个 int 占多少字节? double 呢? bool 呢? 不要使用 sizeof, 除非你要验证自己的答案。
3. 编写一个函数 void to_lower(char* s), 在 C 风格的字符串 s 中用对应的小写字符替换所有大写字符。例如, "Hello, World!" 替换为 "hello, world!"。不要使用任何的标准库函数。C 风格的字符串是一个由 0 结束的字符数组, 因此在结尾你会发现一个值为 0 的字符。
4. 编写一个函数 char* strdup(const char*), 将 C 风格的字符串复制到自由分配的内存中。不要使用任何的标准库函数。

5. 编写一个函数 `char* findx(const char* s, const char* x)`, 在 C 风格的字符串 `s` 中找到字符串 `x` 首次出现的位置。
6. 本章没有说明当你使用 `new` 用尽内存时会发生什么。这种情况称为内存耗尽。请弄清将会发生什么。你两个明确的选择: 查找文档或编写一个用无限循环分配内存但不释放的程序。两者都尝试一下, 估计你在失败之前分配了多少内存?
7. 编写一个程序使用 `cin` 将字符读取到自由存储的数组。读取每个字符直到输入惊叹号(!)为止。不要使用 `std::string` 也不要担心内存耗尽。
8. 重新做练习 7, 这次读取到 `std::string`, 而不是自由存储的数组(`string` 知道如何使用自由存储)。
9. 堆栈采用哪种方式: 向上(趋向高地址)或向下(趋向低地址)? 自由存储最初(在使用 `delete` 之前)是如何生长的? 编写程序来找到答案。
10. 查看你对练习 7 的解决方案。输入是否会使数组溢出? 也就是说, 你是否能输入比分配的数组空间更多的字符(一个严重的错误)? 如果你试图输入超过分配空间的字符时, 将会发生什么合理现象? 查看 `realloc()` 并使用它来扩大你分配的空间。
11. 完成 17.10.1 节中的天神的列表例子并运行它。
12. 为什么我们要定义两个版本的 `find()`?
13. 修改 17.10.1 节中的 `Link` 类以保存 `struct God` 的值。`struct God` 包含 `string` 类型的成员: 姓名、神话、坐骑工具和武器。例如, `God("Zeus", "Greek", "", "lightning")` 和 `God("Odin", "Norse", "Eight-legged flying horse called Sleipner", "")`。编写一个函数 `print_all()`, 每行列出一个天神的属性。添加一个成员函数 `add_ordered()`, 将新的元素放置在正确的位置。使用 `God` 类型的值作为链接, 建立来自三个神话的天神列表; 然后将元素(天神)从这个列表移到相应的列表中, 每个列表对应于一个神话。
14. 是否可以使用单向列表来实现 17.10.1 节中的天神的列表例子? 我们是否可以将 `prev` 成员排除在 `Link` 之外? 为什么我们希望这样做? 哪种例子有利于理解单向链表的使用? 只使用单向链表来重新实现这个例子。

附言

我们可以简单地使用向量, 为什么还要困扰于指针和自由空间这样杂乱的、低层次的东西呢? 好的, 答案是有些人设计和实现了向量与类似的抽象, 而我们希望知道它是如何工作的。由于有些编程语言并不提供等同于指针的功能, 因此就将这个问题留给低层的编程。基本上, 这些语言的程序员将对硬件的直接访问任务交给 C++ 程序员或其他适于低层编程的语言的程序员)。我们最喜欢的理由是简单的, 你实际上不可能说自己了解计算机, 直到你知道软件如何适应硬件的道理。那些不知道指针、内存地址等的人, 经常会对编程语言的特性是如何工作的有一些奇怪的想法, 这种错误的想法会导致代码成为“有趣的糟糕代码”。

第 18 章 向量和数组

“买者自慎!”

——忠告

本章将阐述如何通过下标复制及访问向量。为此，我们将讨论一般的复制技术，并讨论向量类型与数组的低层描述之间的关系起来。本章将阐述数组与指针之间的关系以及其使用引发的一些问题。本章还将讨论对于向量类型必不可少的 5 种操作：构造、默认构造、拷贝构造、拷贝赋值以及析构。

18.1 介绍

为了能够在蓝天中翱翔，一架飞机首先需要在跑道上加速直至它的速度足以摆脱地球的引力。当飞机在跑道上前进时，它就像是一辆特别笨重的、丑陋的大卡车；而一旦飞机飞上了天空，它就会立刻变成一种完全不同的、优雅的、高效的交通工具——这才是它的真正本领。

在本章中，我们首先将学习编程语言的一些特性与技术，以使得我们能够摆脱直接管理底层内存空间所带来的困难和束缚。我们试图实现如下目标：当在编程中产生某些逻辑需求时，我们能够使用某些类型准确地实现逻辑需求的所有功能。而为了实现这一目标，我们首先需要解决很多与计算机访问有关的基本限制，例如：

- 一个对象在内存中的大小是固定的。
- 一个对象存放在内存中的某一特定位置。
- 计算机只为该对象提供了有限的基本操作（如拷贝一个字、将两个字中的值相加，等等）。

本质上来说，上述的这些限制只针对 C++ 的基本数据类型与操作（继承自 C 语言；参见 22.2.5 节和第 27 章）。在第 17 章，我们已经使用过了 `vector` 类型。`vector` 类型能够控制所有对它的元素的访问，并且它提供了一些从用户角度来看“很自然”的操作。

本章将着重介绍拷贝这一概念。这是一个重要的技术问题：对一个对象进行拷贝意味着什么？当拷贝操作发生后，主体与副本之间的独立程度如何？有几种拷贝操作？我们如何指定使用哪种拷贝操作？拷贝操作与其他基本操作（例如初始化和清理）之间有什么关系？

当不能使用高层次的类型（如 `vector` 和 `string`）时，我们会不可避免地需要讨论程序是如何使用内存的。我们需要清楚数组与指针之间的关系、它们的用法以及使用中容易犯的错误，而这些信息对于每一个使用 C++ 或 C 进行编程的人而言是十分重要的。

我们在学习的过程中应留意 `vector` 类型对 `vector` 对象的独特性以及 C++ 在低层类型基础上构建高层类型的方法。然而，在每种语言中，“高层”类型（`string`、`vector`、`list`、`map` 等）是通过相同的计算机原语构建的，并反映了对我们在本章中所描述问题的多种解法。

18.2 拷贝

如第 17 章中所示，我们的 `vector` 类型具有如下形式：

```
class vector {  
    int sz;           // the size  
    double* elem;     // a pointer to the elements
```

```

public:
    vector(int s)                // constructor
        :sz(s), elem(new double[s]) {} // allocates memory
    ~vector()                    // destructor
    { delete[] elem; }          // deallocates memory
    // ...
};

```

让我们试图拷贝其中的一个向量:

```

void f(int n)
{
    vector v(3);                // define a vector of 3 elements
    v.set(2,2.2);               // set v[2] to 2.2
    vector v2 = v;              // what happens here?
    // ...
}

```

理想情况下, 向量对象 v2 成为向量对象 v 的一个副本(即“=”意味着拷贝); 也就是说, 对于所有位于区间 $[0: v.size())$ 内的整数 i 而言, $v2.size() == v.size()$ 且 $v2[i] == v[i]$ 。并且, 对象 v2 与 v 所占用的内存将在函数 f() 结束时被系统收回。上述操作是标准库的 vector 类型所完成的操作, 而非我们的 vector 类型所完成的操作。我们的任务是完善我们的 vector 类型使之能够正确完成上述操作。在此之前, 首先需要清楚我们目前的 vector 类型都实现了什么操作。准确地说, 目前的 vector 类型做错了什么? 如何做错的? 为什么会做错? 一旦找出了错误所在, 那么我们就很可能可以解决这些错误, 并能在以后的编程中避免再犯类似的错误。

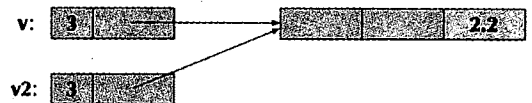
对于一种类型而言, 拷贝的默认含义是“拷贝所有的数据成员”。例如, 我们拷贝一个 Point 对象意味着我们需要拷贝这一对象所包含的坐标数据。但对于指针成员而言, 仅仅对指针成员进行拷贝会产生问题。以例子中的 vector 对象为例, 当拷贝发生后, $v.sz == v2.sz$ 且 $v.elem == v2.elem$ 。因此, 我们的 vector 对象会具有如下形式:

也就是说, 对象 v2 并没有拥有对象 v 的成员副本; 它仅仅共享了 v 的成员。我们可以写出如下代码:

```

v.set(1,99);                // set v[1] to 99
v2.set(0,88);               // set v2[0] to 88
cout << v.get(0) << ' ' << v2.get(1);

```



这段代码将会输出 88 99, 这不是我们想要的结果。如果对象 v 与对象 v2 之间完全独立, 由于我们没有向 $v[0]$ 或 $v[1]$ 写入数据, 因此代码应输出 0 0。你可能会认为我们现在所实现的操作是“有趣的”、“简洁的”或“有用的”, 但这不是我们想要的, 或者不是标准库的 vector 类型所提供的操作。并且, 这些操作将不可避免地会导致函数 f() 返回时发生错误: 对象 v 与 v2 的析构函数将被隐式调用; 对象 v 的析构函数会通过如下语句释放向量元素所占用的内存:

```
delete[] elem;
```

而之后对象 v2 的析构函数也将完成同样的操作。对于对象 v 与 v2 而言, 由于指针 elem 指向同一块内存, 因此重复两次释放这一块内存将会造成灾难性的后果(参见 17.4.6 节)。

18.2.1 拷贝构造函数

那么, 我们应该怎么做呢? 我们需要显示地进行拷贝操作: 当用一个 vector 对象初始化另一个 vector 对象时, 应拷贝所有的向量元素并且保证这一拷贝操作确实被调用了。

某一类型的对象的初始化是由该类型的构造函数实现的。所以, 为实现拷贝操作, 我们需要实现一种特定类型的构造函数。这种类型的构造函数称为拷贝构造函数。C++ 定义拷贝构造函数的参数应该为一个对被拷贝对象的引用。因此, 对于类型 vector 而言, 它的拷贝构造函数为如下形式:

```
vector(const vector&);
```

这一拷贝构造函数将在我们试图使用一个 `vector` 对象初始化另一个 `vector` 对象时被调用。拷贝构造函数使用对象引用作为参数的原因在于我们不想在传递函数参数时又发生参数的拷贝, 而使用 `const` 引用的原因在于我们不想希望函数对参数进行修改(参见 8.5.6 节)。因此, 我们按如下形式重新定义 `vector` 类型:

```
class vector {
    int sz;
    double* elem;
    void copy(const vector& arg);    // copy elements from arg into *elem
public:
    vector(const vector&);           // copy constructor: define copy
    // ...
};
```

函数 `copy()` 简单拷贝参数(一个向量)的所有元素:

```
void vector::copy(const vector& arg)
    // copy elements [0:arg.sz-1]
{
    for (int i = 0; i < arg.sz; ++i) elem[i] = arg.elem[i];
}
```

类型 `vector` 的成员函数 `copy()` 假设在参数对象 `arg` 以及拷贝的目标对象中都包含了 `sz` 个元素。为了保证这种情况的正确性, 我们将函数 `copy()` 实现为类型 `vector` 的私有函数。只有类型 `vector` 的成员函数才能够调用函数 `copy()`, 而这些函数需要确保参数 `arg` 与拷贝的目标对象的大小相匹配。

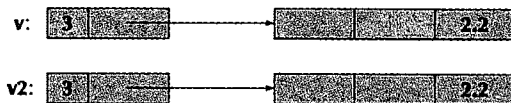
在拷贝参数对象之前, 拷贝构造函数设置了目标对象所包含元素的数目(`sz`)并为元素分配内存(初始化 `elem`):

```
vector::vector(const vector& arg)
    // allocate elements, then initialize them by copying
    :sz(arg.sz), elem(new double[arg.sz])
{
    copy(arg);
}
```

给定这一拷贝构造函数, 再回到我们的例子:

```
vector v2 = v;
```

这一定义语句将会通过如下方式实现对象 `v2` 的初始化: 调用类型 `vector` 的拷贝构造函数并将对象 `v` 作为函数的参数。假设对象 `v` 具有三个元素, 则我们通过拷贝可以得到:



因此, 调用对象 `v` 与 `v2` 的析构函数时不会产生错误, 对象 `v` 与 `v2` 中元素集合所占用的内存都将被正确地释放。可以看出, 拷贝构造函数使对象 `v` 与 `v2` 相互独立, 因此我们可以在不影响其中一个对象的元素的前提下改变另外一个对象的元素。例如:

```
v.set(1,99);    // set v[1] to 99
v2.set(0,88);   // set v2[0] to 88
cout << v.get(0) << ' ' << v2.get(1);
```

这段代码将输出 0 0。

除了通过如下语句之外

```
vector v2 = v;
```

我们还可以通过如下形式实现对象 v2 的初始化

```
vector v2(v);
```

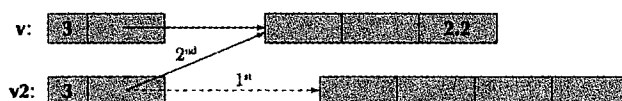
当对象 v(初始化值)与对象 v2(被初始化变量)属于同一类型并且该类型定义了拷贝函数时,则这两种初始化方式完全相同。读者可以选择自己喜欢的方式进行对象初始化。

18.2.2 拷贝赋值

我们可以通过构造函数拷贝(初始化)对象,但我们也可以通过赋值的方式进行 vector 对象的拷贝。与拷贝初始化类似,拷贝赋值默认只进行对象成员的拷贝。因此,对于我们目前定义的 vector 类型而言,拷贝赋值会造成数据重复删除(如 18.2.1 节中的拷贝构造函数所示)以及内存泄漏等问题。例如:

```
void f2(int n)
{
    vector v(3);           // define a vector
    v.set(2,2.2);
    vector v2(4);
    v2 = v;              // assignment: what happens here?
    // ...
}
```

我们希望对象 v2 成为对象 v 的副本(标准库的 vector 类型按这种方式实现),但由于我们并未定义 vector 类型的拷贝赋值操作,因此将执行默认的拷贝赋值操作,即赋值操作将进行成员拷贝。因此,对象 v2 的成员 sz、elem 与对象 v 的成员 sz、elem 完全相同,如下图所示:



当函数 f2() 结束时,程序将会产生与 18.2 节一样的错误:被对象 v 与 v2 共同指向的向量元素将被释放两次(使用 delete[])。另外,还会发生内存泄漏现象:对象 v2 的 4 个元素所占用的内存空间没有被释放。对拷贝赋值操作的改进本质上与对拷贝初始化的改进(见 18.2.1 节)完全相同。我们应按如下方式定义拷贝赋值操作:

```
class vector {
    int sz;
    double* elem;
    void copy(const vector& arg); // copy elements from arg into *elem
public:
    vector& operator=(const vector&); // copy assignment
    // ...
};

vector& vector::operator=(const vector& a)
    // make this vector a copy of a
{
    double* p = new double[a.sz]; // allocate new space
    for (int i = 0; i < a.sz; ++i) p[i] = a.elem[i]; // copy elements
    delete[] elem; // deallocate old space
    elem = p; // now we can reset elem
    sz = a.sz;
    return *this; // return a self-reference (see §17.10)
}
```


由于拷贝赋值操作需要考虑对一个对象原有元素的处理,因此拷贝赋值操作比拷贝构造操作稍微复杂一些。我们的基本策略是从源 vector 对象拷贝所有的元素:

```
double* p = new double[a.sz];    // allocate new space
for (int i = 0; i < a.sz; ++i) p[i] = a.elem[i];
```

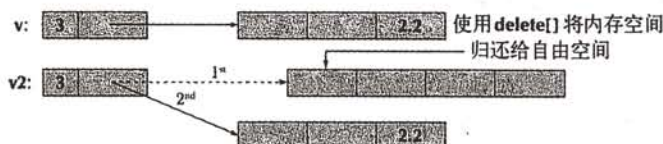
然后,我们将释放目标 vector 对象的原有元素:

```
delete[] elem;    // deallocate old space
```

最后,我们将 elem 指向新元素:

```
elem = p;    // now we can reset elem
sz = a.sz;
```

操作的结果如下图所示:



现在,我们的 vector 类型已不存在内存泄漏以及内存重复释放(delete [])问题了。

在实现拷贝赋值操作时,我们可以在创建副本之前首先释放原有元素所占用的内存以简化代码,但这不是一个好的做法。更好的做法是,我们应一直保留原有元素直到我们确信原有元素能够被安全地释放。如果我们不这么做,那么在将一个对象赋值给它自身时将有可能产生奇怪的结果:

```
vector v(10);
v=v;    // self-assignment
```

请仔细检查我们的实现,以保证它能够正确地处理这种情况(不考虑性能是否最优化)。

18.2.3 拷贝术语

对于大多数的程序以及编程语言而言,拷贝带来了很多问题。一个基本问题是你应该拷贝一个指针(或引用)还是应该拷贝指针指向(或引用)的数据:

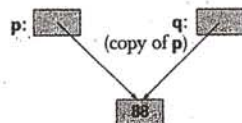
- 浅拷贝只拷贝指针,因此两个指针可能指向同一个对象。
- 深拷贝将拷贝指针指向的数据,因此两个指针将指向两个不同的对象。例如, vector 类型与 string 类型都实现了深拷贝。当我们需要为某一类型的对象实现深拷贝时,我们需要显式地为该类型定义自己的拷贝构造函数与拷贝赋值函数。

下面是一个浅拷贝的例子:

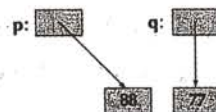
```
int* p = new int(77);
int* q = p;    // copy the pointer p
*p = 88;    // change the value of the int pointed to by p and q
```

浅拷贝的操作如右图所示。与之相对应的是,我们也可以执行深拷贝:

```
int* p = new int(77);
int* q = new int(*p);    // allocate a new int, then copy the value pointed to by p
*p = 88;    // change the value of the int pointed to by p
```



深拷贝的操作如右图所示。通过拷贝术语可以看出,我们原来的 vector 类型产生问题的根源在于它只实现了浅拷贝,而不是拷贝指针 elem 指向的元素。与标准库的 vector 类型相似,改进版的 vector 类型实现了深拷贝:它为元素分配新的内存空间并进行元素的拷贝。实现了浅拷贝的类型(如



指针与引用)称为具有指针语义或引用语义(它们拷贝地址)。实现了深拷贝的类型(如 `string` 类型和 `vector` 类型)称为具有值语义(它们拷贝指向的值)。从用户的角度看来,具有值语义类型的拷贝操作就像没有涉及指针一样——仅仅只有值被拷贝了。也可以说,在进行拷贝操作时,具有值语义的类型表现得就好像它自己是整数类型一样。

18.3 必要的操作

在学习了前几节的知识之后,现在我们可以讨论如何决定一个类型应选择哪些构造函数、该类型是否应定义析构函数、类型是否应定义拷贝赋值函数这些问题了。本节我们将考虑以下 5 种必要的操作:

- 具有一个或多个参数的构造函数
- 默认构造函数
- 拷贝构造函数(拷贝同一类型的对象)
- 拷贝赋值函数(拷贝同一类型的对象)
- 析构函数

通常,我们需要一个或多个构造函数以采用不同的参数实现对象初始化。例如:

```
string s("Triumph");    // initialize s to the character string "Triumph"
vector<double> v(10);    // make v a vector of 10 doubles
```

初始值的含义/用途完全取决于构造函数。标准 `string` 类型的构造函数使用一个字符串作为初始值,而标准 `vector` 类型的构造函数使用一个整数作为向量元素数目的初始值。通常我们使用构造函数来建立不变式(参见 9.4.3 节)。如果我们不能定义一个类的构造函数能够建立的好的不变式,那么很可能我们使用了一个糟糕的类设计或者简单的数据结构。

具有参数的构造函数的形式随它们所在类型的不同而不同。而与之相比,其他的操作则具有更加规则的形式。

我们如何知道一个类型是否需要默认构造函数呢?如果我们希望在不指定初始值的前提下构造该类型的对象,那么该类型就需要默认构造函数。最常见的例子是我们希望将某一类型的对象存放在标准库的 `vector` 对象之中。下面的这些代码是正确的,因为我们为类型 `int`、`string` 和 `vector<int>` 设置了默认值:

```
vector<double> vi(10);    // vector of 10 doubles, each initialized to 0.0
vector<string> vs(10);    // vector of 10 strings, each initialized to ""
vector<vector<int>> vvi(10); // vector of 10 vectors, each initialized to vector()
```

因此,默认构造函数常常是有用的。问题现在变为:“何时拥有一个默认构造函数是有意义的?”一个答案是:“当我们可以为类的不变式设定一个有意义或者显然的默认值的时候”。对于值类型,如 `int` 和 `double`,显然的默认值为 0(对于 `double`,为 0.0)。对于 `string` 类型,默认值为空字符串“”,这也是显然的。对于 `vector` 类型,默认值为空向量。对于类型 `T` 而言,当默认值存在时,则 `T()` 为类型 `T` 的默认值。例如, `double()` 为 0.0、`string()` 为“”, `vector<int>()` 为具有 `int` 类型元素的空向量。

如果一个类型需要获取系统资源,则该类型需要析构函数。由于系统的资源是有限的,因此当我们对资源使用完毕时,应将资源返回给系统。例如,我们可以通过使用 `new` 获得内存资源,而应通过使用 `delete` 或者 `delete[]` 将获得的内存资源释放。我们实现的 `vector` 类型需要获得内存资源以保存它的元素,并在使用完毕后将资源释放,因此 `vector` 类型需要实现析构函数。其他类

型的资源包括文件(如果你打开了一个文件,则你需要负责将它关闭)、锁、线程句柄以及套接字(用于计算机或进程间的通信)。

类型需要析构函数的另一个特征是该类型具有指针成员或引用成员。如果一个类型具有指针成员或引用成员,则该类型通常要实现析构函数以及拷贝操作。

通常,一个实现了析构函数的类型同时也需要实现拷贝构造函数与拷贝赋值函数。其原因很简单,如果该类型的一个对象获取了资源(或者具有指向资源的指针成员),那么只进行默认拷贝(浅拷贝)几乎肯定会带来错误。`vector` 类型就是一个典型的例子。

另外,对于一个基类而言,如果它的派生类具有析构函数,则该基类的析构函数应为虚函数(参见 17.5.2 节)。

18.3.1 显示构造函数

只具有一个参数的构造函数定义了一个从其参数类型向该函数所属类型的转换。这种转换是十分重要的。例如:

```
class complex {
public:
    complex(double);    // defines double-to-complex conversion
    complex(double,double);
    // ...
};

complex z1 = 3.14;    // OK: convert 3.14 to (3.14,0)
complex z2 = complex(1.2, 3.4);
```

尽管如此,我们应谨慎地使用隐式转换,因为隐式转换可能会造成不可预料的后果。例如,我们目前实现的 `vector` 类型定义了一个采用 `int` 类型参数的构造函数。这意味着这一构造函数定义了一个从 `int` 类型向 `vector` 类型的转换。例如:

```
class vector {
    // ...
    vector(int);
    // ...
};

vector v = 10;    // odd: makes a vector of 10 doubles
v = 20;    // eh? Assigns a new vector of 20 doubles to v

void f(const vector&);
f(10);    // eh? Calls f with a new vector of 10 doubles
```

看起来好像我们获得的东西比预料中的更多。幸运的是,我们能够通过一种简单的方式禁止将构造函数用于类型的隐式转换。由关键字 `explicit` 修饰的构造函数(即显式构造函数)只能用于对象的构造而不能用于隐式转换。例如:

```
class vector {
    // ...
    explicit vector(int);
    // ...
};

vector v = 10;    // error: no int-to-vector conversion
v = 20;    // error: no int-to-vector conversion
vector v0(10);    // OK

void f(const vector<double>&);
f(10);    // error: no int-to-vector<double> conversion
f(vector<double>(10));    // OK
```

为了避免意外的类型转换，在我们（以及标准）定义的 `vector` 类型中，具有一个参数的构造函数都是显式构造函数。遗憾的是构造函数默认是非显式的；当我们拿不定主意时，我们最好将所有的具有一个参数的构造函数定义为显式构造函数。

18.3.2 调试构造函数与析构函数

在程序的执行过程中，构造函数与析构函数都将在明确的、可预计的时间点上被调用。尽管如此，我们并不是总是需要采用显式的方式来调用这些函数，如 `vector(2)`。我们在做某些事的时候也会调用这些函数，例如声明一个 `vector` 对象，以传值方式传递一个 `vector` 对象参数，或者使用 `new` 创建 `vector` 对象。构造函数与析构函数的调用可能会造成人们对语法的混淆。下面是常见的构造函数与析构函数被调用的场合：

- 每当类型 `X` 的一个对象被构建时，类型 `X` 的一个构造函数将被调用。
- 每当类型 `X` 的一个对象被销毁时，类型 `X` 的析构函数将被调用。

每当类型的一个对象被销毁时，该类型的析构函数将被调用；这种情况可能发生在变量的作用域结束时、程序结束时或者 `delete` 作用于一个指向对象的指针时。每当类型的一个对象被构建时，该类型的构造函数将被调用；这种情况可能发生在变量被初始化时、通过 `new` 构建对象（除了内建类型）时以及拷贝对象时。

为了对这个问题进行体会，我们在构造函数、赋值函数以及析构函数内加入了打印语句。例如：

```
struct X {           // simple test class
    int val;

    void out(const string& s,int nv)
        { cerr << this << "->" << s << ": " << val << "(" << nv << ")" << "\n"; }

    X(){ out("X()", 0); val=0; }
    X(int v) { out("X(int)", v); val=v; }
    X(const X& x){ out("X(X&)", x.val); val=x.val;}
    X& operator=(const X& a)
        { out("X::operator=", a.val); val=a.val; return *this; }
    ~X() { out("~X()", 0); }
};
```

我们这么做的目的在于使类型 `X` 能够在程序运行中留下一些信息以供我们学习。例如：

```
X glob(2);           // a global variable

X copy(X a) { return a; }

X copy2(X a) { X aa = a; return aa; }

X& ref_to(X& a) { return a; }

X* make(int i) { X a(i); return new X(a); }

struct XX { X a; X b; };

int main()
{
    X loc(4);         // local variable
    X loc2 = loc;
    loc = X(5);
    loc2 = copy(loc);
    loc2 = copy2(loc);
    X loc3(6);
    X& r = ref_to(loc);
```

```

delete make(7);
delete make(8);
vector<X> v(4);
XX loc4;
X* p = new X(9);    // an X on the free store
delete p;
X* pp = new X[5];   // an array of Xs on the free store
delete [] pp;
}

```

试着执行这一程序。

试一试 运行这一程序示例并确保你能够弄清程序结果的含义。如果你这么做了，你就能明白对象的构造与析构的大致过程。

根据你所使用编译器的质量的不同，你可能会发现一些与函数 `copy()` 和 `copy2()` 有关的“丢失拷贝”。我们可以看出这两个函数并没有做任何有意义的事情：它们仅仅将值原封不动地从函数输入拷贝到函数输出。如果编译器的智能足以发现这一事实，那么编译器将会消除函数 `copy()` 和 `copy2()` 对拷贝构造函数的调用。一些较为智能的编译器能够消除不必要的拷贝。

为什么要如此麻烦地设计这样一个类型 `X` 呢？这有点像音乐家们所必须做的指法练习。通过做这些简单的事情，那些真正有意义的事情就比较容易理解了。并且，如果你对构造函数和析构函数存有疑问，那么你也可以在自己实际的类型的构造函数中加入打印语句以了解这些函数的工作过程。对于大一些的程序而言，添加打印语句显得有些繁琐，但技术本质是相似的。例如，你可以通过判断构造函数的调用次数与析构函数的调用次数的差值是否为零，确定程序中是否存在内存泄漏问题。忘记为获取了资源或者具有指针成员的类型定义构造函数与析构函数是十分常见的问题——避免这一问题很容易。

如果你的程序存在很大的问题以至于不能通过这些简单的方法进行处理，那么你应该学会使用一些专业的工具来处理这些问题；这些工具通常称为“内存泄漏探测器”。当然，最理想的情况是通过使用能够避免内存泄漏的技术使得内存泄漏问题不会产生。

18.4 访问向量元素

到目前为止(参见 17.6 节)，我们已经使用过类型 `vector` 的成员函数 `set()` 和 `get()` 访问 `vector` 对象包含的元素，但这些用法比较繁琐、低效。我们希望能够通过下标符号 `v[i]` 对 `vector` 对象中的元素进行访问。为了实现这一目标，我们需要定义一个称为 `operator[]` 的成员函数。下面是我们的初次尝试：

```

class vector {
    int sz;           // the size
    double* elem;     // a pointer to the elements
public:
    // ...
    double operator[](int n) { return elem[n]; } // return element
};

```

上面的代码看起来令人满意且实现简单，但不幸的是实现过于简单了。上面的下标操作 (`operator[]()`) 只实现了对元素的读操作而没有实现对元素的写操作：

```

vector v(10);
int x = v[2];    // fine
v[3] = x;        // error: v[3] is not an lvalue

```

这里 `v[i]` 被解释为函数调用 `v.operator[](i)`，且调用返回对象 `v` 的编号为 `i` 的元素。对于上述

vector 类型而言, `v[3]` 是一个浮点类型的数值, 而不是一个浮点类型的变量。

试一试 编写这一版本的 vector 类型的完整实现, 并观察对于语句 `v[3] = x;`, 编译器将会报告怎样的错误消息。

我们进一步进行如下尝试: 函数 `operator[]` 返回指向对应元素的指针:

```
class vector {
    int sz;           // the size
    double* elem;     // a pointer to the elements
public:
    // ...
    double* operator[](int n) { return &elem[n]; } // return pointer
};
```

根据函数定义, 我们可以编写如下代码:

```
vector v(10);
for (int i=0; i<v.size(); ++i) { // works, but still too ugly
    *v[i] = i;
    cout << *v[i];
}
```

`v[i]` 在这里被解释为函数调用 `v.operator[] (i)`, 且调用返回指向对象 `v` 的编号为 `i` 的元素的指针。但这种实现的问题是, 在我们对元素进行访问时, 我们不得不首先使用操作符“*”对指针进行解引用。这样的实现无疑会使得元素的访问变得繁琐。我们可以通过使下标操作符返回元素的引用以解决这一问题:

```
class vector {
    // ...
    double& operator[] (int n) { return elem[n]; } // return reference
};
```

现在, 我们可以编写如下代码:

```
vector v(10);
for (int i=0; i<v.size(); ++i) { // works!
    v[i] = i; // v[i] returns a reference element i
    cout << v[i];
}
```

上述实现使得对象 `vector` 的下标操作符具有与常规下标操作符相似的含义: `v[i]` 被解释为函数调用 `v.operator[] (i)`, 且调用返回对象 `v` 的编号为 `i` 的元素的引用。

18.4.1 对 const 对象重载运算符

到目前为止, `operator[] ()` 的定义存在一个问题: 它不能用于 `const vector` 类型的对象。例如:

```
void f(const vector& cv)
{
    double d = cv[1]; // error, but should be fine
    cv[1] = 2.0;      // error (as it should be)
}
```

其原因在于函数 `vector::operator[] ()` 可能会潜在地改变 `vector` 对象的元素成员。即使该函数实际上没对元素成员进行修改, 编译器仍会认为这是一个错误, 因为我们没有将这一情况告诉它。解决这一问题的方法是为下标操作再定义一个 `const` 成员函数 (参见 9.7.4 节):

```
class vector {
    // ...
    double& operator[] (int n); // for non-const vectors
    double operator[] (int n) const; // for const vectors
};
```

在 `const` 版本的函数中, 函数只返回 `double` 类型的值, 而不是返回 `double&` 类型的引用。同样我们

也可以返回 `const double&` 类型,但由于一个 `double` 类型的变量占用的内存空间很少,返回这样的变量的引用是没有必要的(参见 8.5.6 节),因此这里函数通过传值的方式返回数据。现在,我们可以编写如下代码:

```
void ff(const vector& cv, vector& v)
{
    double d = cv[1]; // fine (uses the const [])
    cv[1] = 2.0;       // error (uses the const [])
    double d = v[1];   // fine (uses the non-const [])
    v[1] = 2.0;        // fine (uses the non-const [])
}
```

由于我们常常需要通过 `const` 引用的方式传递 `vector` 对象,因此为函数 `operator[]()` 实现 `const` 版本是十分必要的。

18.5 数组

我们已经通过使用数组来引用在自由存储区中顺序排列的对象。与命名变量一样,我们也可以在其他的地方分配数组。实际上,数组可以作为

- 全局变量(但定义全局变量通常是一个糟糕的主意)
- 局部变量(但数组作为局部变量时会受到严格的限制)
- 函数成员(但一个数组不知道其自身大小)
- 类的成员(但数组成员难于初始化)

现在,你可能会发觉我们更赞成使用 `vector` 类型而不是数组。我们应当尽可能地用 `vector` 类型取代数组。尽管如此,数组在 `vector` 对象出现之前就已经存在了很长的时间,并且它与其他编程语言中(如 C 语言)的数组提供的功能大致相同,因此我们必须学会如何使用数组,以便我们能够处理那些很久以前编写的代码,或者那些由不能使用 `vector` 类型的人编写的代码。

那么,什么是数组呢?我们该如何定义数组?我们该如何使用数组?数组是在内存空间中顺序排列的同类型对象的集合;也就是说,数组的所有元素都具有相同的类型,并且各元素之间不存在内存空隙。数组中的元素从 0 开始顺序编号的。数组可以用“方括号”表示:

```
const int max = 100;
int gai[max];           // a global array (of 100 ints); "lives forever"
void f(int n)
{
    char lac[20];        // local array; "lives" until the end of scope
    int lai[60];
    double lad[n];       // error: array size not a constant
    // ...
}
```

注意,数组的使用存在一个限制:对于一个命名数组而言,在程序编译时必须知道该数组包含元素的数目。如果你希望元素的数目是一个变量,那么你必须在自由存储区中分配数组,并通过指针数组进行访问。`vector` 类型就是这么做的。

像在自由存储区存放的数组一样,我们通过下标与解引用操作符(`[]`和`*`)访问命名数组。例如:

```
void f2()
{
    char lac[20];        // local array; "lives" until the end of scope

    lac[7] = 'a';
```

```

    *lac = 'b';           // equivalent to lac[0]='b'

    lac[-2] = 'b';        // huh?
    lac[200] = 'c';       // huh?
}

```

这个函数能够通过编译，但我们知道“通过编译”并不意味着函数能够“正确工作”。操作符[]的使用是显而易见的，但函数并没有进行范围检查。因此，虽然函数 `l2()` 通过了编译，但对 `lac[-2]` 和 `lac[200]` 进行写操作的后果是灾难性的，我们应避免这样的操作。

那么编译器是否能够知道 `lac` 只有 20 个元素以至于能够知道 `lac[200]` 是错误的呢？编译器能够这样做，但就我们所知，到目前为止没有哪一个编译器实现了这样的功能。问题在于在编译期间跟踪数组的范围通常来说是不可能的，并且只查找那些在最简单情况中存在的错误（如上面的错误）并不是十分有用。

18.5.1 指向数组元素的指针

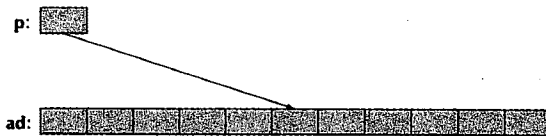
指针可以指向数组的元素。例如：

```

double ad[10];
double* p = &ad[5];    // point to ad[5]

```

现在，指针 `p` 指向 `double` 型元素 `ad[5]`，如下图所示：



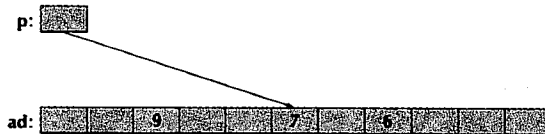
我们能够对指针使用下标与解引用操作符：

```

*p = 7;
p[2] = 6;
p[-3] = 9;

```

我们得到下图：



也就是说，我们可以使用正数或负数作为指针的下标操作数。只要元素位于数组的范围之内，那么这样的操作就是正确的。然而，通过指针访问位于数组范围之外的数据是非法的（参见 17.4.3 节）。通常，编译器不能监测对数组范围之外数据的访问，并且这样的访问很可能是灾难性的。

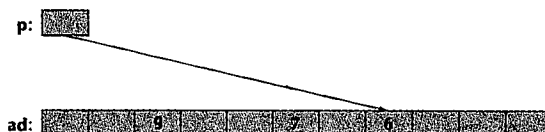
当指针指向一个数组时，加操作与下标操作能够改变指针，使得指针指向数组中的其他元素。例如：

```

p += 2;           // move p 2 elements to the right

```

我们可以得到下图：



并且

```
p -= 5;    // move p 5 elements to the left
```

我们可以得到下图：



通过操作符 `+`、`-`、`+=`、`-=` 移动指针的操作称为指针运算。当进行这种类型的操作时，我们需要保证指针指向的范围，指针只能在数组的范围内进行移动：

```
p += 1000;    // insahe: p points into an array with just 10 elements
double d = *p; // illegal: probably a bad value
               // (definitely an unpredictable value)
*p = 12.34;    // illegal: probably scrambles some unknown data
```

不幸的是，由指针运算所造成的错误有时很难被发现。通常最好的策略是尽量避免使用指针运算。

指针运算最常见的操作是对指针进行自增操作(使用 `++`)以使指针指向下一个元素，以及对指针进行自减操作(使用 `--`)以使指针指向上一个元素。例如，我们可以通过如下方式打印 `ad` 元素的取值：

```
for (double* p = &ad[0]; p < &ad[10]; ++p) cout << *p << '\n';
```

或者反向打印：

```
for (double* p = &ad[9]; p >= &ad[0]; --p) cout << *p << '\n';
```

通过这样的方式进行指针运算是比较常见的。尽管如此，我们发现在编写上面一个例子(反向)时很容易产生错误。为什么是 `&ad[9]` 而不是 `&ad[10]`？为什么是 `>=` 而不是 `>`？这些例子也可以通过下标操作很好地实现。这些例子也可以通过使用 `vector` 类型实现，而 `vector` 类型更容易实现数组范围的检查。

注意，指针元素另一种常用的方式是将指针作为函数的参数进行传递。在这种情况下，编译器并不知道指针指向的数组包含元素的个数：你需要主动地提供这些信息。

为什么 C++ 允许进行指针运算呢？指针运算可能会造成错误，并且我们能够通过下标操作实现指针运算所提供的任何操作。例如：

```
double* p1 = &ad[0];
double* p2 = p1+7;
double* p3 = &p1[7];
if (p2 != p3) cout << "impossible!\n";
```

C++ 允许指针运算主要是因为历史原因。指针运算在很久以前就在 C 语言中存在，将它剔除会造成很多代码不能够运行。还有部分原因在于，在一些低层次的应用(例如内存管理器)中，使用指针运算更为便利。

18.5.2 指针和数组

数组的名字代表了数组的所有元素。例如：

```
char ch[100];
```

`ch` 的大小 `sizeof(ch)` 为 100。然而，数组的名字可以转化(退化)为指针。例如：

```
char* p = ch;
```

`p` 被初始化为 `&ch[0]`，并且 `sizeof(p)` 通常为 4(而非 100)。

这一实现是十分有用的。例如，考虑函数 `strlen()`，它能够统计一个以 0 结尾的字符数组中包含的字符总数：

```
int strlen(const char* p)    // similar to the standard library strlen()
{
    int count = 0;
    while (*p) { ++count; ++p; }
    return count;
}
```

我们可以通过 `strlen(ch)` 或 `strlen(&ch[0])` 对函数进行调用。你可能认为这不过是一个非常小的书写上的优势，我们赞同这一看法。

将数组名转化为指针的一个原因在于避免将大量数据以传值的方式进行参数传递。例如：

```
int strlen(const char a[])    // similar to the standard library strlen()
{
    int count = 0;
    while (a[count]) { ++count; }
    return count;
}

char lots[100000];

void f()
{
    int nchar = strlen(lots);
    // ...
}
```

你可能会认为 `strlen` 函数调用时会将函数参数所指向的 100 000 个字符进行拷贝，但实现上这一操作并不会发生。取而代之的是，编译器会认为参数 `char p[]` 等价于 `char* p`，并且函数调用 `strlen(lots)` 等价于 `strlen(&lots[0])`。这一实现会帮助你避免拷贝操作的庞大开销，但可能会使你感到惊奇。为什么呢？因为在其他的所有情况下，当你向函数以传值方式传递一个对象时，这一对象总会被拷贝。

注意，通过数组名获得的指向数组第一个元素的指针不是一个变量，我们不能对它进行赋值操作：

```
char ac[10];
ac = new char[20];    // error: no assignment to array name
&ac[0] = new char[20]; // error: no assignment to pointer value
```

最终，编译器将会发现这一错误！

作为数组名向指针隐式转换的一个结果，我们不能通过赋值操作拷贝数组：

```
int x[100];
int y[100];
// ...
x = y;    // error
int z[100] = y;    // error
```

如果你需要拷贝一个数组，你必须编写一些更复杂的代码来实现。例如：

```
for (int i=0; i<100; ++i) x[i]=y[i];    // copy 100 ints
memcpy(x,y,100*sizeof(int));    // copy 100*sizeof(int) bytes
copy(y,y+100, x);    // copy 100 ints
```

注意，C 语言不支持像 `vector` 之类的类型，因此在 C 中，你必须使用数组类型。这意味着仍有很多的 C++ 代码使用数组类型。特别地，C 风格的字符串（以 0 结尾的字符数组，参见 27.5 节）是十分常见的。

如果你希望使用赋值操作实现数组拷贝，那么你应该使用像 `vector` 之类的类型。等价于上述拷贝代码的 `vector` 实现为：


```
vector<int> x(100);
vector<int> y(100);
// ...
x = y;    // copy 100 ints
```

18.5.3 数组初始化

与 vector 以及用户定义的其他容器相比,数组具有一个十分重要的优点: C++ 提供数组初始化的支持。例如:

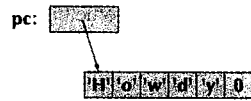
```
char ac[] = "Beorn";    // array of 6 chars
```

数一数上面的字符数,共有 5 个,但 ac 是一个具有 6 个元素的数组:编译器会自动在字符串的末尾添加字符 0,如右图所示。字符串以 0 结尾在 C 以及很多系统中都作为字符串的规范。我们称以 0 结尾的字符串数组为 C 风格的字符串。所有的字符串常量都是 C 风格的字符串。例如:

```
char* pc = "Howdy";    // pc points to an array of 6 chars
```

右图描绘了这一风格。注意,以 0 为取值的字符不等于字符 '0' 或其他字母数字。以 0 结尾的目的在于帮助函数定位字符的结尾。应记住的是:数组并不知道它本身的实际大小。根据以 0 结尾这一规范,我们可以编写如下代码:

```
int strlen(const char* p)    // similar to the standard library strlen()
{
    int n = 0;
    while (p[n]) ++n;
    return n;
}
```



实际上,我们不需要自己实现 strlen() 这一函数,因为该函数是一个在头文件 <string.h> 中定义的标准库函数(参见 27.5 节和附录 B10.3)。注意, strlen() 只统计字符数,并不统计字符串结尾的 0 字符;也就是说,我们需要 $n+1$ 个字符空间以存储 C 风格字符串中包含的 n 个字符。

只有字符数组能够通过字符串进行初始化,但所有的数组都能够通过与其类型相匹配的值列表进行初始化。例如:

```
int ai[] = { 1, 2, 3, 4, 5, 6 };    // array of 6 ints
int ai2[100] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };    // the last 90 elements are initialized to 0
double ad[100] = { };    // all elements initialized to 0.0
char chars[] = { 'a', 'b', 'c' };    // no terminating 0!
```

注意, ai 的元素个数为 6(不是 7)且 chars 的元素个数为 3(而非 4)——“在末尾加 0”这一规则只适用于字符串。如果在初始化数组时未明确指定其大小,那么编译器将通过初始化列表中包含的元素个数确定数组的大小。这是相当有用的特性。如果初始化的元素个数小于数组的实际大小(如 ai2 和 ad),数组的剩下元素将被设置为该元素类型的默认值。

18.5.4 指针问题

与数组类似,指针在使用过程中常会出现问题。因此,我们将在本节对经常出现的问题进行总结。特别地,由指针所产生的严重问题通常涉及对数据的意外访问,并且很多这一类型的问题都涉及对数组范围之外的数据的访问。在本节中,我们主要考虑如下问题:

- 使用 null(空)指针进行数据访问。
- 使用非初始化指针进行数据访问。
- 对数组结尾之后的数据进行访问。

- 对未分配对象的访问。
- 对作用域之外的对象进行访问。

在所有的情况下，对于编程者而言，一个实际的问题是所有的访问看起来都没有问题；指针仅仅是没有被赋予合适的值。更糟糕的是，这些问题可能会在某些表面上不相关的对象出现错误之后才会显现。例如：

不要使用 null 指针进行数据访问：

```
int* p = 0;
*p = 7;    // ouch!
```

在实际的程序中，当问题发生时，在指针的初始化与使用之间通常还存在一些其他的代码。特别地，向函数传递 p 或者使用函数的返回值作为 p 的取值是十分常见的例子。我们通常不应该向函数传递 null 指针，但如果我们不得不这么做时，我们应在使用指针前测试指针是否为 null：

```
int* p = fct_that_can_return_a_0();
if (p == 0) {
    // do something
}
else {
    // use p
    *p = 7;
}
```

并且

```
void fct_that_can_receive_a_0(int* p)
{
    if (p == 0) {
        // do something
    }
    else {
        // use p
        *p = 7;
    }
}
```

使用引用(参见 17.9.1 节)和使用异常以捕获错误(参见 5.6 节和 19.5 节)是避免空指针的主要工具。

对指针进行初始化：

```
int* p;
*p = 9;    // ouch!
```

特别地，不要忘了对作为类成员的指针进行初始化。

不要访问不存在的数组元素：

```
int a[10];
int* p = &a[10];
*p = 11;    // ouch!
a[10] = 12; // ouch!
```

在一个循环中访问第一个元素以及最后一个元素时应特别小心。取而代之的是，我们可以使用 vector 类型。如果我们确实需要在多于一个的函数中使用数组(将它作为函数的参数)，那么我们就应极其小心并且应将数组的大小也作为函数的参数进行传递。

不要通过一个已经进行 delete 操作的指针访问数据：

```
int* p = new int(7);
// ...
delete p;
// ...
*p = 13;    // ouch!
```

代码 `delete p` 以及此后的代码会改写 `p` 指向的内存区域中包含的数据。在所有问题中,我们认为这一问题是最难发现和避免的。防止这一问题的最有效方法是避免出现“暴露的”`new` 操作以及 `delete` 操作: 只在构造函数与析构函数中使用 `new` 与 `delete`, 或者使用容器(如 `Vector_ref`(参见附录 E.4))来处理 `delete`。

不要返回指向局部变量的指针:

```
int* f()
{
    int x = 7;
    // ...
    return &x;
}

// ...

int* p = f();
// ...
*p = 15; // ouch!
```

函数 `f()` 的返回值以及此后的代码会造成对 `p` 指向内存区域中数据的改写。造成这一问题的原因在于, 函数中的局部变量在进入函数时被分配内存空间(在栈中), 而局部变量所占用的内存空间将在函数退出时被释放。特别地, 在类对象中的局部变量所占内存空间将在类的析构函数调用时被释放(参见 17.5.1 节)。编译器不能够监测与返回指向局部变量指针相关的大部分问题。

例如下面的一个在逻辑上等价的例子:

```
vector& ff()
{
    vector x(7);
    // ...
    return x;
} // the vector x is destroyed here

// ...

vector& p = ff();
// ...
p[4] = 15; // ouch!
```

很少的编译器能够监测这一变量的返回问题。

程序员们很可能会低估这些问题。然而, 很多有经验的程序员都曾被这些问题打败。解决的方法是在代码中随意使用指针、数组、`new` 和 `delete`。如果你这么做了, 那么在实际的程序中, 光靠小心谨慎是不够的。取而代之的是, 我们应使用 `vector`、`RAII`(Resource Acquisition Is Initialization; 参见 19.5 节), 以及其他的系统途径管理内存和其他资源。

18.6 实例: 回文

我们已经展示了足够多的技术上的例子! 让我们尝试一个难题。回文是一种单词, 它按照顺序以及逆序的方式拼写所得的结果是一致的。例如, `anna`、`petep` 以及 `malayalam` 都是回文, 而 `ida` 和 `homesick` 不是回文。有两种方法判断一个单词是否是回文:

- 获得单词按逆序拼写所得的单词副本, 并将其与原有单词进行比较。
- 判断单词的首字符与尾字符是否相同, 然后判断第二个字符与倒数第二个字符是否相同, 持续进行这一操作直到到达单词的中央。

这里我们将采取第二种方法。根据单词的表示方式以及跟踪字符比较进度的方式的不同，我们可以通过多种方式实现第二种方法。我们将采用不同的方法实现回文的判断，以观察不同的语言特征是如何影响代码的形式和工作方式的。

18.6.1 使用 string 实现回文

首先，我们使用标准库的 `string` 类型以及 `int` 类型的索引跟踪字符比较的进度：

```
bool is_palindrome(const string& s)
{
    int first = 0;           // index of first letter
    int last = s.length()-1; // index of last letter
    while (first < last) {    // we haven't reached the middle
        if (s[first]!=s[last]) return false;
        ++first;             // move forward
        --last;              // move backward
    }
    return true;
}
```

当比较到达单词的中央且未发现不同的字符时，函数返回 `true`。我们建议，在你编写这段代码时，应保证代码在下列情况下都是正确的：当字符串不包含字符时；当字符串只包含一个字符时；当字符串包含奇数或偶数个字符时。当然，我们可以不只是依靠逻辑来判断代码是否正确。我们可以按照下面的方式测试：

```
int main()
{
    string s;
    while (cin>>s) {
        cout << s << " is";
        if (!is_palindrome(s)) cout << " not";
        cout << " a palindrome\n";
    }
}
```

我们使用 `string` 类型的原因在于“`string` 类型更便于处理单词”。通过 `string` 类型能够很容易地将以空白字符分隔的单词读入字符串，并且一个 `string` 对象清楚它的实际大小。如果我们希望通过 `is_palindrome()` 判断包含空白字符的字符串是否为回文，那么我们可以使用函数 `getline()`（见 11.5 节）。这时，函数结果会认为 `ah ha` 和 `as df fd sa` 为回文。

18.6.2 使用数组实现回文

如果不能使用 `string` 类型（或 `vector` 类型）时，那么我们只好使用数组存储字符。例如：

```
bool is_palindrome(const char s[], int n)
    // s points to the first character of an array of n characters
{
    int first = 0;           // index of first letter
    int last = n-1;          // index of last letter
    while (first < last) {    // we haven't reached the middle
        if (s[first]!=s[last]) return false;
        ++first;             // move forward
        --last;              // move backward
    }
    return true;
}
```

为了测试 `is_palindrome()`，我们首先需要将字符读入数组。为了实现这一操作，一种安全的方法如下：

```
istream& read_word(istream& is, char* buffer, int max)
// read at most max-1 characters from is into buffer
{
    is.width(max);    // read at most max-1 characters in the next >>
    is >> buffer;    // read whitespace-terminated word,
                    // add zero after the last character read into buffer
    return is;
}
```

合理地设置对象 `istream` 的宽度能够避免 `>>` 操作导致缓冲区溢出。不幸的是,这意味着我们不知道读操作是以空白符为结束,还是以缓冲区满为结束(因此我们还需要继续读入更多的字符)。另一方面,谁还记得 `width()` 函数在作为输入时表现出的具体行为信息? 标准库中的 `string` 与 `vector` 类型更适合于作为输入的缓冲区,因为它们能够根据输入数据的规模动态调整它们的大小。结束字符 `0` 是必需的,因为对字符数组(C 风格字符串)的大多数操作总是假设字符串以 `0` 结束。通过函数 `read_word()`,我们可以编写如下代码:

```
int main()
{
    const int max = 128;
    char s[max];
    while (read_word(cin,s,max)) {
        cout << s << " is";
        if (!is_palindrome(s,strlen(s))) cout << " not";
        cout << " a palindrome\n";
    }
}
```

函数 `strlen(s)` 返回当函数调用 `read_word()` 结束之后,数组 `s` 中包含的字符总数。并且 `cout << s` 将输出数组中的元素,直至遇见字符 `0` 为止。

使用数组的代码实现要远比使用 `string` 类型的代码实现复杂,并且当我们试图处理长字符串时,这种情况会变得更糟。参见习题 10。

18.6.3 使用指针实现回文

除了使用索引,我们还可以通过指针识别字符:

```
bool is_palindrome(const char* first, const char* last)
// first points to the first letter, last to the last letter
{
    while (first < last) {    // we haven't reached the middle
        if (*first != *last) return false;
        ++first;    // move forward
        --last;    // move backward
    }
    return true;
}
```

注意,实际上我们可以对指针进行自增操作或自减操作。自增操作使指针指向数组中的下一个元素,而自减操作使指针指向上一个元素。如果指针指向的区域超出了数组的实际范围,那么将会产生严重的越界错误。这是使用指针可能会产生的问题。

我们通过如下方式调用 `is_palindrome()`:

```
int main()
{
    const int max = 128;
    char s[max];
    while (read_word(cin,s,max)) {
        cout << s << " is";
    }
}
```

```

        if (!is_palindrome(&s[0], &s[strlen(s)-1])) cout << " not";
        cout << " a palindrome\n";
    }
}

```

我们还可以按如下方式重写 `is_palindrome()` :

```

bool is_palindrome(const char* first, const char* last)
// first points to the first letter, last to the last letter
{
    if (first < last) {
        if (*first != *last) return false;
        return is_palindrome(++first, --last);
    }
    return true;
}

```

当我们重新描述回文的定义时, 上述代码的意义就显而易见了: 一个单词是回文, 当它的首字符与尾字符相同, 并且删除首字符与尾字符所形成的单词子串仍然是回文。



简单练习

本章有两个练习: 一个涉及数组, 另一个涉及 `vector` 类型, 且两个练习的内容大致相同。读者可以分别完成这两个练习, 并对练习进行比较。

数组练习:

1. 定义大小为 10 的 `int` 类型的全局数组 `ga`, 并将数组元素初始化为 1, 2, 4, 8, 16 等。
2. 定义具有两个参数的函数 `f()`, 其中一个参数为 `int` 类型数组, 另一个参数指出该数组包含的元素数量。
3. 在函数 `f()` 中:
 - a) 定义大小为 10 的 `int` 类型的局部数组。
 - b) 将 `ga` 的值拷贝至 `la`。
 - c) 打印 `la` 的所有元素。
 - d) 定义整型指针 `p`, 并将其初始化指向一个在自由存储区中分配存储空间的数组, 该数组与参数数组大小相同。
 - e) 将参数数组包含的值拷贝至在自由存储区中分配的数组。
 - f) 打印在自由存储区中分配的数组的所有元素。
 - g) 释放在自由存储区中分配的数组所占的空间。
4. 在函数 `main()` 中:
 - a) 调用 `f()` 并以 `ga` 为其参数。
 - b) 定义大小为 10 的数组 `aa`, 并将其元素初始化为前 10 个阶乘数(1, 2*1, 3*2*1, 4*3*2*1 等)。
 - c) 调用 `f()` 并以 `aa` 为其参数。

标准库 `vector` 练习:

1. 定义全局变量 `vector<int> gv`, 并将其元素初始化为 1, 2, 4, 8, 16 等。
2. 定义函数 `f()`, 且函数参数为 `vector<int>` 类型的向量。
3. 在函数 `f()` 中:
 - a) 定义局部变量 `vector<int> lv`, 且 `lv` 的大小与参数向量相同。
 - b) 将 `gv` 的值拷贝至 `lv`。
 - c) 打印 `lv` 的所有元素。
 - d) 定义局部变量 `vector<int> lv2`, 并以参数向量初始化 `lv2`。
 - e) 打印 `lv2` 的所有元素。
4. 在函数 `main()` 中:
 - a) 调用 `f()` 并以 `gv` 为其参数。
 - b) 定义向量 `vector<int> vv`, 并将其元素初始化为前 10 个阶乘数(1, 2*1, 3*2*1, 4*3*2*1 等)。

c) 调用 `f()` 并以 `vv` 为其参数。

思考题

1. “买者自慎!”的含义是什么?
2. 对于类对象而言, 拷贝的默认含义是什么?
3. 类对象拷贝的默认含义在什么情况下是合适的? 什么情况下是不合适的?
4. 什么是拷贝构造函数?
5. 什么是拷贝赋值?
6. 拷贝赋值与拷贝初始化之间有什么区别?
7. 什么是浅拷贝? 什么是深拷贝?
8. `vector` 对象的副本与该 `vector` 对象之间的比较如何?
9. 类的 5 种必要操作有哪些?
10. 什么是显式构造函数? 在什么情况下应该使用显式构造函数?
11. 对于一个类对象而言, 哪些操作是被隐式调用的?
12. 什么是数组?
13. 如何拷贝一个数组?
14. 如何对数组初始化?
15. 什么时候应该使用指针参数而不是引用参数? 为什么?
16. 什么是 C 风格的字符串?
17. 什么是回文?

术语

数组	深拷贝	显式构造函数	数组初始化	默认构造函数	回文
拷贝赋值	必要操作	浅拷贝	拷贝构造函数		

习题

1. 编写函数 `char* strdup(const char*)`, 该函数能够将 C 风格的字符串复制至其在自由存储区中分配的内存当中。要求: 不使用任何标准库函数。使用解引用操作符“`*`”代替下标操作。
2. 编写函数 `char* findx(const char* s, const char* x)`, 该函数能够在 C 风格字符串 `s` 中定位字符串 `x` 首次出现的位置。要求: 不使用任何标准库函数。使用解引用操作符“`*`”代替下标操作。
3. 编写函数 `int strcmp(const char* s1, const char* s2)`, 该函数能够比较 C 风格的字符串。如果 `s1` 按照字典顺序排在 `s2` 之前, 函数返回负整数; 如果 `s1` 与 `s2` 相同, 函数返回 0; 如果 `s1` 按照字典顺序排在 `s2` 之后, 函数返回正整数。要求: 不使用任何标准库函数。使用解引用操作符“`*`”代替下标操作。
4. 考虑如下问题: 如果向函数 `strdup()`、`findx()` 与 `strcmp()` 传递非 C 风格字符串的参数, 会出现什么结果? 试一试。首先, 尝试向函数传递不以 0 为结尾的字符数组(不要在实际(非实验性)代码中编写这样的代码; 否则可能会造成严重的破坏)。尝试一下传递在自由存储区中以及栈中分配的“虚假 C 风格字符串”。如果程序结果看上去仍然是合理的, 那么请关闭 `debug` 模式。重新设计并实现这三个函数, 使得这些函数具有另一个参数, 该参数指定了字符串参数中包含的最大元素个数。然后, 通过正确的 C 风格字符串以及“坏”字符串测试这些函数。
5. 编写函数 `string cat_dot(const string& s1, const string& s2)`, 该函数将参数字符串以圆点相连接。例如, `cat_dot(“Niels”, “Bohr”)` 返回结果 `Niels. Bohr`。
6. 改写习题 5 的函数 `cat_dot()`, 使函数具有第三个字符串参数, 且该参数指定分隔字符串(而非圆点)。
7. 改写习题 6 的函数 `cat_dot()`, 使函数以 C 风格字符串为参数, 并以在自由存储空间中分配的字符串作为返回结果。要求: 不使用任何标准库函数或类型。保证所有在自由存储空间中分配(通过 `new`)的内存空间都被正确地释放(通过 `delete`)。

8. 改写 18.6 节中的所有函数。函数通过构造单词的逆向副本并将副本与原有单词相比较的方式确定单词是否为回文。例如，由“home”产生“emoh”，然后比较两个单词是否相同。因此，home 不是一个回文。
 9. 考虑 17.3 节中的内存布局。编写一个程序，该程序能够指出：静态存储区、栈区、自由存储区在内存中的布局顺序；栈扩展的方向：是向高地址空间扩展还是向低地址空间扩展？在自由存储区中分配的数组中，具有较高下标的元素是在高地址空间存储中还是低地址空间中存储？
 10. 回顾 18.6.2 节中关于回文问题的数组解决方案。修改该方案使之能够对长字符串进行处理：当输入字符串过长时，会提示出错报告；能够处理任意长度的字符串。评价这两种实现版本的复杂度。
 11. 查找（例如，通过网页）skip list 的概念并实现这种类型的列表。
 12. 编程实现游戏“猎杀怪兽”。“猎杀怪兽”是一种由 Gregory Yob 发明的简单电脑游戏。它的基本前提是一个满身臭味的怪物居住在一个由多间相连房间构成的黑暗的山洞里。你的任务是通过弓箭杀死怪兽。除了怪兽之外，山洞中还存在两种危险：无底陷阱和巨型蝙蝠。如果你进入的房间有无底陷阱，那么游戏将结束。如果房间中有蝙蝠，那么蝙蝠将把你抓入另一房间之中。如果房间中有怪兽或者怪兽进入了你所在的房间，它会吃掉你。当你进入一个房间时，你将会得到一个关于附近是否存在危险的提示：“我闻到了怪兽的味道”：是指怪兽在相邻的房间中。
“我感到一阵微风吹来”：是指相邻的一间房间中有陷阱。
“我听到蝙蝠的声音”：是指相邻的房间中有蝙蝠。
- 山洞中的每一个房间都编了号。每一个房间通过地道均与其他三个房间相连。当进入一个房间时，你将会得到诸如“你在房间 12 中；房间 12 通过地道与房间 1、13、4 相连；移动还是射击？”一种可能的答案是 m13（代表移动到房间 13）以及 s13-4-3（向房间 13、4、3 射箭）。一支箭的覆盖范围为三个房间。在游戏开始时，你有 5 支箭。射击的后果是它会惊醒怪兽并且怪兽将向与它当前所在房间相邻的房间逃跑——可能是你所在的房间。
- 这一练习的难点可能在于决定房间的相连关系。你可能需要使用随机数生成器（例如 `std_lib_facilities.h` 中声明的 `randint()` 函数）产生不同的山洞。提示：在调试中，应使程序能够产生关于山洞状态的输出。

附言

标准库 `vector` 类型建立在低层次的内存管理工具基础之上，例如指针与数组，而它的主要功能是帮助我们避免使用这些工具所带来的复杂性。当我们设计一个类时，我们必须考虑类的初始化、拷贝与析构。

第 19 章 向量、模板和异常

“成功不是终点”

——Winston Churchill

本章将讨论最常见、最有用的 STL 容器的设计与实现：vector。在本章中，我们将展示如何实现元素数量可变的容器，如何以参数形式指定容器中元素的类型，如何处理越界错误。与前面类似，本章中介绍的技术是通用的，而不仅仅局限于 vector 类型的实现，甚至不仅仅局限于容器的实现。对于各种不同的数据类型，我们将展示如何安全地处理数量可变的不同类型的数据。此外，我们还会介绍一些实际问题，作为设计上的经验教训。本章中所用技术依赖于模板与异常，所以我们将阐述如何定义模板，另外，针对如何使用好异常这一问题，我们将介绍一些用于资源管理的基本技术。

19.1 问题

在第 18 章结束时，我们设计的 vector 类型已经可以实现如下功能：

- 创建 vector 类型对象，其元素类型为双精度浮点类型且元素数量可以任意设置。
- 通过赋值与初始化拷贝 vector 对象。
- 通过 vector 本身，在它们离作用域时，正确地释放所占用的内存空间。
- 通过传统的下标操作访问 vector 对象中的元素（可以在赋值操作符 = 的左边和右边）。

掌握这些知识是有用的，但为了进一步加深对 vector 的了解（根据我们使用标准库中 vector 类型的经验），我们需要解决下面三个问题：

- 如何改变 vector 对象的大小（改变其包含元素的个数）？
- 如何捕获和报告对 vector 元素的访问越界？
- 如何以参数的方式指定 vector 元素的类型？

例如，如何定义 vector 使得下面的操作是合法的：

```
vector<double> vd;           // elements of type double
double d;
while(cin>>d) vd.push_back(d); // grow vd to hold all the elements

vector<char> vc(100);        // elements of type char
int n;
cin>>n;
vc.resize(n);                // make vc have n elements
```

显然，使 vector 类型能够完成上述操作是十分有用的，但从编程的角度看，为什么这些操作是重要的呢？对于一个单一实体 vector，我们可以改变 vector 的两种属性：

- 元素的数量
- 元素的类型

可变性是十分有用的。在日常生活中，我们总是要收集数据。看看我的桌子，我看见了一堆银行声明、信用卡账单、电话话费单。而这些东西本质上是一系列的各种类型的数据信息的集合：字符串以及数值。在我的面前是一部电话；它记录了联系人的姓名与电话号码。在房间的书架上，满是书籍。我们的程序也是相似的：程序中包含具有各种不同元素类型的容器。程序中我

们需要使用不同类型的容器(`vector` 仅仅是最常用的一种), 且它们包含诸如电话号码、姓名、交易总额等信息。本质上来说, 关于我的桌子和房间的例子都源于某些计算机程序。唯一的例外是电话: 电话可以类比为—台计算机, 当我们查找联系人电话号码时, 我们实际上看到的是一个程序的输出——该程序与我们现在编写的程序类似。实际上, 这些号码都被存放在一个 `vector<Number>` 类型的对象之中。

显然, 并不是所有的容器都具有相同的元素数量。那么我们是否可以只使用元素数量固定的容器呢? 也就是说, 在编写代码时是否不使用诸如 `push_back()`、`resize()` 之类的操作? 答案是肯定的。但是这将会给我们带来不必要的负担: 在程序中只使用固定大小的容器的基本技巧是, 当我们需要增加容器包含的元素数量时, 我们需要首先将原有容器包含的元素移至一个更大的容器之中。例如:

```
// read elements into a vector without using push_back:
vector<double>* p = new vector<double>(10);
int n = 0;      // number of elements
double d;
while(cin >> d) {
    if (n==p->size()) {
        vector<double>* q = new vector<double>(p->size()*2);
        copy(p->begin(), p->end(), q->begin());
        delete p;
        p = q;
    }
    (*p)[n] = d;
    ++n;
}
```

这并不完美。你能说服我们这些代码是正确的吗? 你确认吗? 注意, 在上述代码中, 我们是如何突然开始使用指针并显式地进行内存管理的? 我们在上述代码中所使用的内存管理技术是针对大小固定的对象(数组; 参见 18.5 节)的。然而, 使用容器(如 `vector`)的一个重要的原因是, 它能够处理元素数量的动态变化, 从而帮助我们避免麻烦并减少出错的机会。换句话说, 容器能够根据用户的实际需要动态调整其大小。例如:

```
vector<double>vd;
double d;
while(cin>>d) vd.push_back(d);
```

在编程中, 容器大小的变化是否是常见操作呢? 如果不是, 那么容器大小的动态改变并不会带来多少便利。然而, 在实际中, 容器大小变化是十分常见的操作。最明显的例子是通过容器从输入中读取未知数量的数值。其他例子包括从一次查找中收集结果集合(我们事先不知道结果的总数量)以及从数据集合中依次删除元素。因此, 我们所面临的问题不是我们是否应该处理容器大小的变化, 而是如何处理这样的变化。

为什么我们需要做这些事情呢? 为什么不“一次性分配足够的内存”? 这看起来是一种最简单且最有效率的策略。然而, 这一策略能够实现, 仅当我们能够根据需要可靠地分配足够的内存, 并避免由于分配的内存过多造成浪费——实际上这是不可能的。如果我们采用了这种策略, 我们将不得不重写代码(如果我们认真并系统地检查溢出错误)或者疲于应付由这种策略带来的麻烦(如果我们在检查方面比较粗心的话)。

显然, 并不是所有的 `vector` 具有相同类型的元素。我们需要使用 `vector` 存储 `double` 类型的数值、温度数据、记录(各种类型)、字符串、操作、GUI 按钮、形状、日期、窗口指针等。

容器的类型多种多样。这一点非常重要, 由于它有一些重要的含义, 因此不应该不加思

考就接受它。除了 `vector`，我们为什么还需要其他类型的容器呢？如果我们只使用一种类型的容器（如 `vector`），我们就只需要将所有精力用于实现该容器，并可以将它作为编程语言的一部分实现。如果我们只使用一种容器，我们就不必学习其他类型的容器。我们可以总是使用 `vector`。

对于大多数重要的应用而言，数据结构是十分关键的。大量的书籍阐述了应该如何组织数据，而这些书籍包含的知识可以概括为对这一问题的回答：“我们怎样才能最好地存储自己的数据？”因此，我们需要使用不同类型的容器。到目前为止，我们已经使用过 `vector` 和 `string` 类型（`string` 是存储字符的容器）。在第 20 章，我们还将学习 `list`、`map`（`map` 用于存储数值对）以及矩阵等类型。由于我们需要使用各种类型的容器，因此学习用于建立并使用容器的编程语言特征及编程技术是极为有用的。

在内存管理的最底层，所有的对象都具有固定的大小并且不存在类型的概念。在这一章中，我们将介绍用于实现容器大小动态改变的语言特征与编程技术。

19.2 改变向量大小

为实现容器大小的动态改变，标准库的 `vector` 类型采用了什么方法呢？它提供了三种简单的操作。假设我们定义

```
vector<double> v(n); // v.size()==n
```

我们可以通过三种方法改变 `v` 的大小：

```
v.resize(10);           // v now has 10 elements

v.push_back(7);        // add an element with the value 7 to the end of v
                        // v.size() increases by 1

v = v2;                // assign another vector; v is now a copy of v2
                        // v.size() now equals v2.size()
```

针对容器大小的动态改变，标准库的 `vector` 类型提供了更多的操作，如 `erase()` 和 `insert()`（附录 B.4.7），但在本章中我们只考虑如何在我们的 `vector` 类型中实现这三种操作。

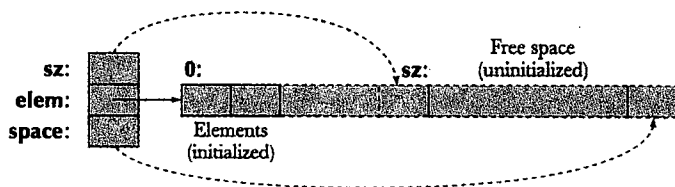
19.2.1 方法描述

在 19.1 节中，我们介绍了实现容器大小改变的最简单的策略：为新的元素数量分配存储空间，并将原有元素拷贝至新的存储空间。然而，如果我们需要经常调整容器大小，这一策略将是低效的。在实际中，如果我们曾经改变了容器的大小，那么以后我们很可能还需要多次进行这样的操作。例如，我们很少只进行一次 `push_back()` 操作。因此，针对这一实际情况，我们可以对程序进行优化。实际上，所有的对 `vector` 类型的实现都会记录 `vector` 对象中实际包含元素的数量以及 `vector` 对象为“将来的扩展”所保留的空闲存储空间的总量。例如：

```
class vector {
    int sz;           // number of elements
    double* elem;    // address of first element
    int space;       // number of elements plus "free space"/"slots"
                    // for new elements ("the current allocation")

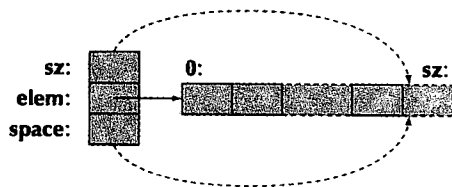
public:
    // ...
};
```

上述实现可以由下图表示：



由于我们从 0 开始统计元素的数量, 那么 `sz` (元素数量) 标示了最后一个元素之后的位置, 而 `space` 标示了最后一个存储单元之后的位置。而图中的指针则标示了 `elem + sz` 以及 `elem + space` 的位置。

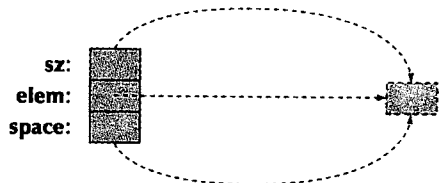
当 `vector` 对象被构造时, `space` 为 0, 如下图所示。在这里, 我们不会分配额外的内存空间, 除非我们需要开始改变元素的数量。 `space == sz` 意味着此时的 `vector` 并没有额外的内存, 除非我们使用 `push_back()` 操作。



默认构造函数 (构造不包含任何元素的 `vector` 对象) 将这三个成员变量设为 0:

```
vector::vector() :sz(0), elem(0), space(0) { }
```

也就得到右面的图。图中所示的存储单元是虚构的, 实际中并不存在。默认构造函数并不为元素分配内存, 且只占用最小的存储空间 (参见习题 16)。



请注意, 我们的 `vector` 类型所采用的技术可以用于实现标准的向量 (或其他数据结构), 但在我们的系统中, 标准库的实现 `std::vector` 可能采用了不同的技术。

19.2.2 reserve 和 capacity

用于改变 `vector` 大小的最基本的操作 (即改变元素的数量) 是 `vector::reserve()`。这一操作可用于为新的元素分配内存空间:

```
void vector::reserve(int newalloc)
{
    if (newalloc <= space) return;           // never decrease allocation
    double* p = new double[newalloc];       // allocate new space
    for (int i=0; i<sz; ++i) p[i] = elem[i]; // copy old elements
    delete[] elem;                         // deallocate old space
    elem = p;
    space = newalloc;
}
```

注意, 我们并不对保留空间中的元素进行初始化。我们只是保留存储空间以备将来使用; 使用保留空间是 `push_back()` 和 `resize()` 的工作。

显然, `vector` 对象中保留的空闲空间的大小对于用户可能是有用的, 因此我们 (与标准库中的 `vector` 类型类似) 提供了一个函数以获得这一信息:

```
int vector::capacity() const { return space; }
```

也就是说, 对于一个 `vector` 对象 `v`, `v.capacity() - v.size()` 代表了在不重新分配存储空间的前提下, 我们通过 `push_back()` 操作能够向 `v` 添加元素的数量。

19.2.3 resize

实现了函数 `reserve()` 后, `resize()` 的实现是十分简单的。我们只需要处理以下几种情况:

- 容器新的大小大于容器原有的分配空间。
- 容器新的大小大于容器当前的大小，但小于或等于容器的原有分配空间。
- 容器新的大小等于容器当前的大小。
- 容器新的大小小于容器当前的大小。

下面的代码展示了 `resize()` 函数的实现：

```
void vector::resize(int newsize)
    // make the vector have newsize elements
    // initialize each new element with the default value 0.0
{
    reserve(newsize);
    for (int i=sz; i<newsize; ++i) elem[i] = 0;    // initialize new elements
    sz = newsize;
}
```

我们使用函数 `reserve()` 处理内存空间的管理。代码中的循环将初始化新的元素（如果有）。

在这里，我们不显式地处理每一种情况。但你可以验证：在上述代码中，每一种情况均被正确地处理了。

试一试 如果我们想要证明上述 `resize()` 是否正确，那么我们需要考虑（并测试）哪些情况？当 `newsize == 0` 时会怎样？当 `newsize == -77` 呢？

19.2.4 push_back

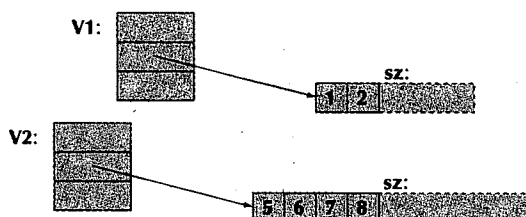
表面上看来，`push_back()` 是复杂的，难于实现。实际上，当实现了函数 `reserve()` 之后，`push_back()` 的实现是相当简单的：

```
void vector::push_back(double d)
    // increase vector size by one; initialize the new element with d
{
    if (space==0) reserve(8);           // start with space for 8 elements
    else if (sz==space) reserve(2*space); // get more space
    elem[sz] = d;                       // add d at end
    ++sz;                               // increase the size (sz is the number of elements)
}
```

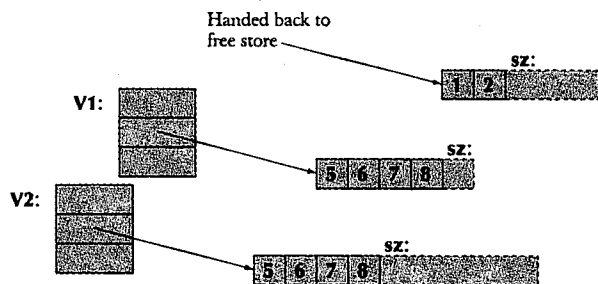
换句话说，当 `vector` 对象没有空闲的内存空间以容纳新元素时，我们将已有的内存空间扩大一倍。在实际中，这种内存空间的扩大策略对于绝大多数的 `vector` 应用是一个很好的选择，并且这种策略已被用于大多数的标准库 `vector` 类型的实现。

19.2.5 赋值

我们可以采用几种不同的方法实现向量的赋值。例如，仅当赋值涉及的两个向量具有相同的元素数量时，我们才可以判定赋值是合法的。然而，在 18.2.2 节中，我们认为向量赋值应具有更为通用的意义：当赋值 `v1 = v2` 完成后，向量 `v1` 应是向量 `v2` 的副本。例如：



显然，我们需要拷贝元素，那么向量对象所包含的空闲存储空间呢？我们是否需要拷贝空闲存储空间？我们不需要：新的 `vector` 对象将会获得元素的副本，但由于我们不确定此后新 `vector` 对象将被如何使用，因此我们不会对对象的空闲存储空间进行处理。如下图所示：



最简单的实现包括如下操作：

- 为副本分配存储空间。
- 拷贝元素。
- 删除原有存储空间。
- 更新 `sz`、`elem`、`space` 的值。

例如：

```
vector& vector::operator=(const vector& a)
// like copy constructor, but we must deal with old elements
{
    double* p = new double[a.sz];           // allocate new space
    for (int i = 0; i < a.sz; ++i) p[i] = a.elem[i]; // copy elements
    delete[] elem;                          // deallocate old space
    space = sz = a.sz;                      // set new size
    elem = p;                               // set new elements
    return *this;                           // return self-reference
}
```

作为惯例，赋值操作符将返回关于被赋值对象的引用。符号 `*this` 的含义参见 17.10 节。

上述实现是正确的，但通过观察我们可以发现上述实现包含了大量多余的存储空间的分配与释放操作。如果被赋值对象的大小大于赋值对象的大小时会如何？如果被赋值对象的大小等于赋值对象的大小时会如何？在很多应用中，后一种情况是十分常见的。在这两种情况中，我们仅仅只需要将元素拷贝至目标 `vector` 对象：

```
vector& vector::operator=(const vector& a)
{
    if (this==&a) return *this; // self-assignment, no work needed

    if (a.sz <= space) {           // enough space, no need for new allocation
        for (int i = 0; i < a.sz; ++i) elem[i] = a.elem[i]; // copy elements
        sz = a.sz;
        return *this;
    }

    double* p = new double[a.sz]; // allocate new space
    for (int i = 0; i < a.sz; ++i) p[i] = a.elem[i]; // copy elements
    delete[] elem;                          // deallocate old space
    space = sz = a.sz;                      // set new size
    elem = p;                               // set new elements
    return *this;                           // return a self-reference
}
```

在上述代码中，我们首先测试自引用的情况（如 `v = v`）；在这种情况下，我们不需要做任何操作。这一测试在逻辑上看是多余的，但有时会带来性能的优化。上述代码展示了在成员函数中通过 `this` 指针判断参数 `a` 是否与当前对象是同一对象的一种通用方法。实际上，如果我们删除了

this == &a, 代码仍然能够正确地工作。另外, a.sz <= space 这一判断也是为了代码的优化, 如果我们删除了 a.sz <= space, 代码仍然能够正确地工作。

19.2.6 到现在为止我们设计的 vector 类

到目前为止, 我们差不多已经完成了 double 向量类型的设计:

```
// an almost real vector of doubles:
class vector {
/*
    invariant:
        for 0<=n<sz elem[n] is element n
        sz<=space;
        if sz<space there is space for (space-sz) doubles after elem[sz-1]
*/
    int sz;           // the size
    double* elem;     // pointer to the elements (or 0)
    int space;        // number of elements plus number of free slots
public:
    vector() : sz(0), elem(0), space(0) { }
    explicit vector(int s) : sz(s), elem(new double[s]), space(s)
    {
        for (int i=0; i<sz; ++i) elem[i]=0; // elements are initialized
    }

    vector(const vector&);           // copy constructor
    vector& operator=(const vector&); // copy assignment

    ~vector() { delete[] elem; }     // destructor

    double& operator[](int n) { return elem[n]; } // access
    const double& operator[](int n) const { return elem[n]; }

    int size() const { return sz; }
    int capacity() const { return space; }

    void resize(int newsize);        // growth
    void push_back(double d);
    void reserve(int newalloc);
};
```

请注意上述代码是如何实现一些必备的操作的(参见 18.3 节): 构造函数、默认构造函数、拷贝操作、析构函数。上述 vector 类型实现了元素访问操作(通过下标[]), 实现了获取数据信息的操作(size()与 capacity())以及实现了控制容器大小的操作(resize()、push_back()和 reserve())。

19.3 模板

但在实际编程中, 我们可能不仅仅需要 double 向量; 我们希望能够自由地指定 vector 类型所包含的元素类型。例如:

```
vector<double>
vector<int>
vector<Month>
vector<Window*>           // vector of pointers to Windows
vector<vector<Record>>    // vector of vectors of Records
vector<char>
```

为了达到目的, 我们必须知道如何定义模板。我们从第一天开始就已经会使用模板了, 但到目前为止我们还从未定义一个模板。标准库提供了需要使用的工具, 但我们仍然需要清楚标准库

的设计者们是如何实现诸如 `vector` 类型和 `sort()` 函数这些工具的(参见 21.1 节和附录 B.5.4)。对我们来说,这不仅仅是理论上的兴趣,因为(通常)标准库所采用的工具和技术对于我们编写自己的代码而言是十分有用的。例如,在第 21 和第 22 章中,我们将展示模板是如何被用于实现标准库容器和算法的。在第 24 章,我们将展示如何为科学计算设计矩阵类型。

本质上说,模板是一种机制,它使程序员能够使用数据类型作为一个类或函数的参数。当我们将数据类型作为参数时,编译器将会根据参数生成特定的类或函数。

19.3.1 类型作为模板参数

我们希望使 `vector` 类型所包含的元素类型能够成为 `vector` 类型的参数。因此,在我们实现的 `vector` 类型中,用 `T` 取代 `double`, 其中 `T` 是一个参数且它能被赋予诸如 `double`、`int`、`string`、`vector`、`<Record>` 和 `Window*` 之类的“值”。在 C++ 中,引入类型参数 `T` 的符号为 `template < class T >` 前缀,这一符号的含义是“对于所有类型 `T`”。例如:

```
// an almost real vector of Ts:
template<class T> class vector {    // read "for all types T" (just like in math)
    int sz;           // the size
    T* elem;          // a pointer to the elements
    int space;         // size+free_space
public:
    vector() : sz(0), elem(0), space(0) { }
    explicit vector(int s);

    vector(const vector&);           // copy constructor
    vector& operator=(const vector&); // copy assignment

    ~vector() { delete[] elem; }     // destructor

    T& operator[](int n) { return elem[n]; } // access: return reference
    const T& operator[](int n) const { return elem[n]; }

    int size() const { return sz; } // the current size
    int capacity() const { return space; }

    void resize(int newsize);        // growth
    void push_back(const T& d);
    void reserve(int newalloc);
};
```

上述代码中,我们将在 19.2.6 节实现的 `double` 的 `vector` 类型中的 `double` 类型替换为模板参数 `T`。我们可以通过如下方式使用类模板 `vector`:

```
vector<double> vd;           // T is double
vector<int> vi;              // T is int
vector<double*> vpd;         // T is double*
vector< vector<int> > vvi;   // T is vector<int>, in which T is int
```

当我们使用模板时,我们可以认为编译器是按照如下方式产生类的:编译器用实际类型取代模板参数。例如,当编译器遇见代码中的 `vector < char >` 时,它将产生如下代码:

```
class vector_char {
    int sz;           // the size
    char* elem;       // a pointer to the elements
    int space;        // size+free_space
public:
    vector_char();
    explicit vector_char(int s);
```



```

vector_char(const vector_char&);           // copy constructor
vector_char& operator=(const vector_char &); // copy assignment

~vector_char ();                          // destructor

char& operator[] (int n);                  // access: return reference
const char& operator[] (int n) const;

int size() const;                          // the current size
int capacity() const;

void resize(int newsize);                  // growth
void push_back(const char& d);
void reserve(int newalloc);
};

```

对于 `vector < double >`，编译器产生的代码与 19.2.6 节中的代码类似（其中将使用一个合适的内部名称表示 `vector < double >`）。

我们有时称类模板为类型生成器，称通过指定类模板的模板参数的方式生成类型（类）的过程为特例或模板实例化。例如，`vector < char >` 和 `vector < Poly_line * >` 称为 `vector` 的特例。在简单的情况下，如 `vector` 类型，实例化是一个简单的过程。而在最普遍、最高级的情况下，模板实例化是一个相当复杂的过程。幸运的是，模板实例化的复杂性是编译器设计者而不是模板使用者需要解决的问题。模板实例化（生成模板特例）只占用程序的编译时间或链接时间，而不会占用程序运行时间。

自然，我们也可以使用类模板的成员函数。例如：

```

void fct(vector<string>& v)
{
    int n = v.size();
    v.push_back("Norah");
    // ...
}

```

当使用类模板的成员函数时，编译器将生成合适的函数。例如，当编译器遇见 `v.push_back("Norah")` 时，它会根据模板定义如下：

```

void vector<string>::push_back(const string& d) { /* ... */ }

```

生成函数

```

template<class T> void vector<T>::push_back(const T& d) { /* ... */ };

```

这样，就得到一个函数以实现 `v.push_back("Norah")` 调用。换句话说，当你需要使用函数处理某一特定类型的参数时，编译器将会根据模板为你生成一个函数。

你可以在模板中使用 `template < typename T >` 代替 `template < class T >`。两者完全相同，但有些人喜欢 `typename`，“因为它的含义更清楚”并且“因为没人会被 `typename` 这一名称迷惑，没人会认为不能使用内建类型（如 `int`）作为模板参数”。我们认为 `class` 这一名称已经包含了类型的含义，因此使用 `class` 并没有什么不好。而且，关键字 `class` 要更短些。

19.3.2 泛型编程

在 C++ 中，模板是泛型编程的基础。实际上，C++ 中“泛型编程”的定义就是“使用模板”，虽然这样的定义有点太单纯了。我们不应根据编程语言的特征定义基本的编程概念。编程语言的特征主要用于支持编程技术——而不是相反的。和其他的热门概念一样，“泛型编程”存在多种定义。我们认为简单的、最有用的定义是：

泛型编程：编写能够正确处理各种不同数据类型参数的代码，只要参数的数据类型满足特定的语法和语义要求。

例如，vector 的元素必须是能够实现拷贝的数据类型（通过拷贝构造和拷贝赋值实现）。在第 20 和 21 章，我们将介绍要求参数支持算术运算的模板。当我们参数化的是一个类时，就获得一个类模板，通常也称为参数化的类型或者参数化的类。当我们参数化的是一个函数时，我们将获得函数模板，通常也称为参数化的函数，有时也称为算法。因此，泛型编程有时称为“面向算法的编程”；其设计的重点在于算法的实现而非算法所使用的数据类型。

由于参数化数据类型的概念是编程的核心，我们需要进一步探讨这个有些让人困惑的术语。这样，当我们在其他场合中再次碰见这一概念时，才不会感到困惑。

泛型编程依赖于显式模板参数的形式通常称为参数化多态。相反，从类层次结构与虚函数中获得的多态称为专用多态，而这一类型的编程称为面向对象的编程（见 14.3 和 14.4 节）。之所以两类编程都称为多态是因为每种类型都依赖于程序员通过一个单一的接口表示一个概念的多个版本。多态在希腊语中是“多种形状”的意思，这意味着你可以通过一个通用的接口就能操纵多个不同的数据类型。在第 16 ~ 19 章的 Shape 例子中，我们可以通过接口 Shape 访问多种形状（如 Text、Circle 和 Polygon）。当我们使用 vector 时，通过定义为 vector 模板的接口使用不同类型的 vector（如 `vector<int>`、`vector<double>` 和 `vector<Shape*>`）。

面向对象的编程（使用类层次结构和虚拟函数）和泛型编程（使用模板）之间存在几个差异。最明显的差异是，在一般编程当中，被调用函数的选择由编译器在编译时确定，而在面向对象的编程当中，函数的选择在程序运行过程中确定。例如：

```
v.push_back(x);    // put x into the vector v
s.draw();         // draw the shape s
```

对于 `v.push_back(x)`，编译器将决定 `v` 的元素类型并采用对应的 `push_back()`；而对于 `s.draw()`，编译器将会间接调用某一 `draw()` 函数（使用 `s` 的 `vtbl`，参见 14.3.1 节）。这一差异使得面向对象的编程比泛型编程更为灵活，但泛型编程更为规则，更容易理解，能被更好地执行（因此使用单词“专用”和“参数化”进行区分）。

让我们对上述内容进行总结：

- 泛型编程：以模板为基础，依靠程序编译时的解析。
- 面向对象的编程：以类层次结构和虚函数为基础，依靠程序运行时的解析。

将这两种类型的编程相结合是可能的也是有用的。例如：

```
void draw_all(vector<Shape*>& v)
{
    for (int i=0; i<v.size(); ++i) v[i]->draw();
}
```

在上述代码中，我们调用了基类 Shape 的一个虚函数（`draw()`）——这是面向对象的编程。然而，我们将 `Shape*` 类型的指针存放在一个 vector 对象之中，`Shape*` 是参数化类型，因此我们在进行泛型编程。

所以（假设你的头脑中现在充满了哲学思想）人们为什么使用模板？为了无与伦比的灵活性和性能，

- 在对性能要求高的场合中使用模板（例如，数值计算和硬实时；参见第 24 和 25 章）。
- 在需要灵活地组合不同类型信息的场合中使用模板（如 C++ 标准库；参见第 20 和 21 章）。

模板具有很多有用的特性，例如高灵活性和近似最优的性能，但不幸的是它们并不是最优的。与其他方法一样，模板也有对应的缺点。对于模板，主要的问题在于模板所带来的灵活性和性能是以模板内部（模板的定义）和模板接口（模板的声明）的分离作为代价的。模板所带来的拙劣的出错诊断证明了这一问题的存在——程序可能会显示大量拙劣的出错信息。通常，在编译过程中，这些出错信息出现的时间要比我们希望的更晚。

当编译一个使用模板的代码时，编译器将查看模板以及模板的参数类型。编译器这么做的目的在于获得足够的信息以生成优化代码。为了使这些信息可用，编译器试图要求模板必须在它被使用的位置完全定义，包括模板的所有成员函数以及在这些成员函数中调用的所有模板函数。因此，模板的设计者会试图将模板的定义放置在头文件之中。虽然 C++ 标准并不要求必须这么做，但我们建议读者这么做：对于将在多个翻译单元中使用的模板，它的定义应包含在头文件之中。

开始时，你可以只编写十分简单的模板并小心地逐步改进以获得相关经验。一种有用的开发技术是：就像我们编写 `vector` 类型一样，首先，编写并测试使用某一特定数据类型的类。当这一步成功之后，用模板参数替代代码中的特定数据类型。出于一般性、类型安全性以及性能考虑，我们应使用基于模板的库，如 C++ 标准库。第 20 和 21 章将会介绍标准库中的容器和算法，并通过例子介绍模板的使用方法。

19.3.3 容器和继承

人们总是尝试采用下面这种结合了面向对象编程和泛型编程的方式：将包含派生类对象的容器以包含基类对象的容器的形式使用。然而，这一方式是错误的。例如：

```
vector<Shape> vs;
vector<Circle> vc;
vs = vc;           // error: vector<Shape> required
void f(vector<Shape>&);
f(vc);             // error: vector<Shape> required
```

但是为什么这一方式是错误的呢？也许你会说我能够将一个 `Circle` 对象转化为一个 `Shape` 对象。实际上，你不能这么做。你可以将 `Circle*` 转化为 `Shape*`，或者将 `Circle&` 转化为 `Shape&`，但我们已经有意地避免了 `Shape` 对象之间的赋值，因此你可能想知道当你将一个拥有半径的 `Circle` 对象赋值给一个不具有半径属性的 `Shape` 变量时，将会发生什么（参见 14.2.4 节）。假如我们允许它发生，将会发生所谓的“截断”现象，与整数截断（参见 3.9.2 节）本质上等价的对象截断。

因此，我们试图通过采用指针修改上述代码，如下所示：

```
vector<Shape*> vps;
vector<Circle*> vpc;
vps = vpc;           // error: vector<Shape*> required
void f(vector<Shape*>&);
f(vpc);              // error: vector<Shape*> required
```

这一次，系统仍然报错；为什么呢？看一看 `f()` 可能会做些什么：

```
void f(vector<Shape*>& v)
{
    v.push_back(new Rectangle(Point(0,0),Point(100,100)));
}
```

显然，我们可以将一个 `Rectangle*` 指针放入 `vector<Shape*>`。但是，如果这个 `vector<Shape*>` 对象在别的地方被解释为一个 `vector<Circle*>` 对象时，可能会得到出人意料的糟糕结果。特别地，假设上述例子能够在编译器中成功编译，那么在 `vpc` 中存放 `Rectangle*` 指针会产生什么结果呢？继承是一种强大且微妙的机制，而模板并没有向继承扩展。有几种方法能够通过使用模板表

示继承,但这些内容不在本书的介绍范围之内。我们只需要记住,对于任何模板 C 而言,“D 是 B”并不意味着“ $C < D >$ 是 $C < B >$ ”——并且我们通过这一准则避免偶尔的类型违例。参见 25.4.4 节。

19.3.4 整数作为模板参数

显然,通过类型使类参数化是有用的。那么,通过“其他东西”使类参数化又如何呢,如整数值和字符串值?本质上来说,任何参数种类都是有用的,但在本节中我们只考虑以数据类型和整数作为参数。其他的参数种类并不像这两种参数那么有用,并且在 C++ 中,在使用其他参数类别前通常需要了解关于它们的语言特征的细节。

下面讨论一个使用整数值作为模板参数的最常见的例子:一个容器所包含的元素数量在编译时已经被确定:

```
template<class T, int N> struct array {
    T elem[N];                // hold elements in member array

    // rely on the default constructors, destructor, and assignment

    T& operator[] (int n);      // access: return reference
    const T& operator[] (int n) const;

    T* data() { return elem; } // conversion to T*
    const T* data() const { return elem; }

    int size() const { return N; }
};
```

我们能够按如下方式使用上述 array(参见 20.7 节):

```
array<int,256> gb;           // 256 integers
array<double,6> ad = { 0.0, 1.1, 2.2, 3.3, 4.4, 5.5 }; // note the initializer!
const int max = 1024;

void some_fct(int n)
{
    array<char,max> loc;
    array<char,n> oops; // error: the value of n not known to compiler
    // ...
    array<char,max> loc2 = loc; // make backup copy
    // ...
    loc = loc2;                // restore
    // ...
}
```

显然, array 是很简单的(比 vector 更简单但实现的功能更有限)。那么,为什么人们希望使用 array 而不是 vector 呢?一种答案是“效率”。我们在编译时就已经知道 array 的大小,因此编译器将会分配静态内存(为全局对象,如 gb)和栈内存(为局部变量,如 loc)而不是在自由空间中分配内存空间。当我们进行范围检查时,检查可根据已知常数(参数 N)进行。对于大多数程序而言,效率的提高并不是十分重要的,但如果我们编写的是一个关键的系统组件,例如网络驱动,那么程序的效率就变得十分重要了。更重要的是,有些程序可能不允许使用自由空间。这类程序通常属于嵌入式系统的程序和/或安全性要求严格的程序(参见第 25 章)。在这类程序中, array 在处理临界限制时(不使用自由空间)要比 vector 更具有优势。

现在考虑一个相反的问题:不是“为什么我们不只使用 vector”而是“为什么不使用内建的数组”?如我们在 18.5 节中所见,使用数组容易造成错误:数组不知道它们自身的大小,它们能够

很容易地转化为指针，它们不能进行彼此间的直接拷贝；而 array 不存在这些问题。例如：

```
double* p = ad;           // error: no implicit conversion to pointer
double* q = ad.data();    // OK: explicit conversion
```

```
template<class C> void printout(const C& c)
{
    for (int i = 0; i < c.size(); ++i) cout << c[i] << '\n';
}
```

与 vector 类型类似，我们能够对 array 调用函数 printout()：

```
printout(ad);             // call with array
vector<int> vi;
// ...
printout(vi);             // call with vector
```

这是一个将泛型编程应用于数据访问的简单例子。这段代码能够正确运行的原因在于 array 和 vector 类型的接口 (size() 和下标操作) 是相同的。第 20 和 21 章将会详细介绍这种编程风格。

19.3.5 模板参数推导

对于类模板，当你生成某一特定类的对象时，你需要指定模板参数。例如：

```
array<char,1024> buf;      // for buf, T is char and N is 1024
array<double,10> b2;       // for b2, T is double and N is 10
```

对于函数模板，编译器通常能够从函数的参数中推断出模板参数。例如：

```
template<class T, int N> void fill(array<T,N>& b, const T& val)
{
    for (int i = 0; i < N; ++i) b[i] = val;
}

void f()
{
    fill(buf, 'x');        // for fill(), T is char and N is 1024
                           // because that's what buf has
    fill(b2,0.0);          // for fill(), T is double and N is 10
                           // because that's what b2 has
}
```

在技术上，fill(buf, 'x') 是 fill < char, 1024 > (buf, 'x') 的简写，fill(b2, 0) 是 fill < double, 10 > (b2, 0) 的简写。幸运的是，我们并不需要按照这一规定编写代码。编译器能够自动为我们做这些事情。

19.3.6 一般化 vector 类

当我们将“包含 double 型元素的 vector”类推广至“包含 T 类型元素的 vector”模板时，我们并没有重新考虑函数 push_back()、resize() 和 reserve() 的定义。现在，我们必须做这些事情了，因为在 19.2.2 节和 19.2.3 节中，这些函数的实现采用了一些假设，而这些假设对 double 元素类型是成立的，但并不是对所有其他的元素类型都是成立的：

- 如果 X 类型没有默认值，应如何处理 vector < X >？
- 当 vector 对象使用完毕时，如何确保对象的元素被销毁了？

我们必须得解决这些问题吗？我们可能会说，“不要用不具有默认值的类型设置 vector 的元素类型”以及“不要将 vector 用于带有可能产生问题的析构函数的类型”。对于一个以“通用”为目标的工具，上述限制对用户而言是相当苦恼的，并且会使用户产生这样的印象：工具的设计者没有全面考虑问题或者没有考虑用户的需求。通常，这样的猜疑是正确的，但是在标准库的设计者实现的代码中并不存在这些问题。为了构造与标准库 vector 相同的类型，必须解决这些问题。

对于不具备默认值的类型，我们设置了一个选项，已经在我们需要默认值时能够指定该类型的默认值：

```
template<class T> void vector<T>::resize(int newsize, T def = T());
```

也就是说，除非用户指定了默认值，否则使用 `T()` 作为默认值。例如：

```
vector<double> v1;
v1.resize(100);           // add 100 copies of double(), that is, 0.0
v1.resize(200, 0.0);      // add 100 copies of 0.0 — mentioning 0.0 is redundant
v1.resize(300, 1.0);      // add 100 copies of 1.0

struct No_default {
    No_default(int);       // the only constructor for No_default
    // ...
};

vector<No_default> v2(10); // error: tries to make 10 No_default()s
vector<No_default> v3;
v3.resize(100, No_default(2)); // add 100 copies of No_default(2)
v3.resize(200);               // error: tries to make 100 No_default()s
```

析构函数的问题要更难于解决。实际上，我们需要处理这样的数据结构：同时包含已被初始化数据和未被初始化数据的数据结构。到目前为止，我们已经知道如何避免未初始化数据以及由此产生的问题。现在（作为 `vector` 的实现者）我们不得不面对这一问题，使得我们（`vector` 的用户）不需要在实际应用中自己解决这些问题。

首先，我们需要寻找一种获得并管理未初始化内存空间的方法。幸运的是，标准库为我们提供了 `allocator` 类，该类能够提供未初始化内存。下面代码给出了 `allocator` 的一个简化版本：

```
template<class T> class allocator {
public:
    // ...
    T* allocate(int n);           // allocate space for n objects of type T
    void deallocate(T* p, int n); // deallocate n objects of type T starting at p

    void construct(T* p, const T& v); // construct a T with the value v in p
    void destroy(T* p);               // destroy the T in p
};
```

如果你想了解细节信息，那么请参考《The C++ Programming Language》中的 `<memory>`（参见附录 B.1.1）中的内容，或者参考 C++ 标准。尽管如此，上面所列的 4 个基本操作使我们能够实现：

- 分配能够容纳类型 `T` 的一个对象的未初始化内存空间。
- 在未初始化空间中构造类型 `T` 的对象。
- 销毁类型 `T` 的对象，并将其所占内存设置为未初始化状态。
- 释放能够容纳类型 `T` 的一个对象的未初始化内存空间。

`allocator` 正是我们实现 `vector<T>::reserve()` 需要使用的工具。我们可以向 `vector` 传递一个分配器参数：

```
template<class T, class A = allocator<T>> class vector {
    A alloc; // use allocate to handle memory for elements
    // ...
};
```

除了提供分配器（并默认使用标准的分配器而不是使用 `new`），一切与前面的 `reserve()` 实现完全相同。作为 `vector` 的用户，我们可以忽略分配器，直至 `vector` 能够按照一种与众不同的方法管理其元素占用的内存空间。作为 `vector` 的实现者、试图理解基本问题和基本技术的学习者，我们必须明白 `vector` 对象是如何处理未初始化内存并为其用户构造合适的对象的。与之相关的代码是

vector 中直接处理内存的成员函数, 如 `vector<T>::reserve()`:

```
template<class T, class A>
void vector<T,A>::reserve(int newalloc)
{
    if (newalloc<=space) return;           // never decrease allocation
    T* p = alloc.allocate(newalloc);       // allocate new space
    for (int i=0; i<sz; ++i) alloc.construct(&p[i],elem[i]); // copy
    for (int i=0; i<sz; ++i) alloc.destroy(&elem[i]);         // destroy
    alloc.deallocate(elem,space);          // deallocate old space
    elem = p;
    space = newalloc;
}
```

我们将元素拷贝至未初始化的内存空间形成元素的新副本, 并销毁原有的元素。因为对于诸如 `string` 之类的类型, 赋值总是假设目标空间已被初始化, 因此在这种情况下我们不能使用赋值。

实现了 `reserve()` 之后, `vector<T, A>::push_back()` 就容易实现了:

```
template<class T, class A>
void vector<T,A>::push_back(const T& val)
{
    if (space==0) reserve(8);              // start with space for 8 elements
    else if (sz==space) reserve(2*space);   // get more space
    alloc.construct(&elem[sz],val);         // add val at end
    ++sz;                                   // increase the size
}
```

类似地, `vector<T, A>::resize()` 也不难实现:

```
template<class T, class A>
void vector<T,A>::resize(int newsize, T val = T())
{
    reserve(newsize);
    for (int i=sz; i<newsize; ++i) alloc.construct(&elem[i],val); // construct
    for (int i = newsize; i<sz; ++i) alloc.destroy(&elem[i]);      // destroy
    sz = newsize;
}
```

注意, 由于有些类型没有默认的构造函数, 因此需要提供一个选项以使我们能够为元素指定一个默认值。

当我们试图减小当前的 `vector` 时, 在这种情况下, 我们还需要考虑“多余元素”的析构函数。析构函数可以被认为能够实现将一个具有类型的对象转变为“原始内存”的工具。

在代码中“融入分配器”是 C++ 编程的高级知识, 可以将这一内容放在一边, 除非我们对 C++ 已有足够的了解。

19.4 范围检查和异常

回顾我们目前实现的 `vector`, 我们会发现我们没有对数据访问进行范围检查。`Operator[]` 的实现十分简单:

```
template<class T, class A> T& vector<T,A>::operator[](int n)
{
    return elem[n];
}
```

所以, 下面这些代码存在错误:

```
vector<int> v(100);
v[-200] = v[200];    // oops!
int i;
cin>>i;
v[i] = 999;           // maul an arbitrary memory location
```

上述代码能够通过编译运行,并能够访问不属于 `vector` 对象的内存空间。这可能会造成严重的问题!在实际程序中,这样的代码是不可接受的。下面我们将完善 `vector` 以处理这些问题。最简单的方法是增加一个用于访问检查的操作 `at()`:

```
struct out_of_range { /* ... */};    // class used to report range access errors

template<class T, class A = allocator<T>> class vector {
    // ...
    T& at(int n);                      // checked access
    const T& at(int n) const;          // checked access

    T& operator[](int n);              // unchecked access
    const T& operator[](int n) const;  // unchecked access
    // ...
};

template<class T, class A> T& vector<T,A>::at(int n)
{
    if (n<0 || sz<=n) throw out_of_range();
    return elem[n];
}

template<class T, class A> T& vector<T,A>::operator[](int n) // as before
{
    return elem[n];
}
```

当实现了 `at()` 操作后,我们可以编写如下代码:

```
void print_some(vector<int>& v)
{
    int i = -1;
    cin >> i;
    while(i != -1) try {
        cout << "v[" << i << "]==" << v.at(i) << "\n";
    }
    catch(out_of_range) {
        cout << "bad index: " << i << "\n";
    }
}
```

在上面的代码中,我们通过 `at()` 进行数据访问的范围检查,并且我们捕获 `out_of_range` 以避免非法的数据访问。

数据访问通常的做法是,当确定元素索引是有效的时,我们通过 `[]` 使用下标操作;而当元素索引可能造成越界时,我们应使用函数 `at()` 进行数据访问。

19.4.1 附加讨论:设计上的考虑

到目前为止,我们的实现一切顺利,但为什么不在 `operator[]()` 中实现范围检查呢?与我们的实现类似,标准库 `vector` 在 `at()` 中提供了范围检查而在 `operator[]()` 中没有进行范围检查。在本节中,我们将解释一下为什么要这样做。之所以这么做主要存在以下 4 个方面的观点:

- 1) 兼容性:在 C++ 具有异常捕获功能之前,人们就已经在使用不包含范围检查的下标操作。
- 2) 效率:你可以在一个不进行范围检查但性能更优化的操作符基础上实现一个进行范围检查的操作符,但你不能在一个进行范围检查的操作符基础上实现性能更优化的操作符。
- 3) 约束:在一些环境中,异常是不可接受的。

4) 检查的可选性: C++ 标准并没有规定你不能对 `vector` 进行范围检查, 所以如果你希望进行检查, 你应该选择能够进行范围检查的实现。

19.4.1.1 兼容性

人们总是希望他们以前的代码能够正常运行。例如, 如果你编写了一百万行的代码, 为了能在这些代码中正确使用异常而重新编写这些代码是一个浩大的工程。我们可能会认为完善这些代码是有意义的, 但代码的拥有者却不会这么想。而且, 已有代码的维护人员会认为没有进行范围检查的代码可能是不安全的, 但实际上他们的代码已经被测试以及使用了若干年了, 并且所有的程序错误都已经被发现了。我们可以怀疑这一观点, 但没有人能够那么自信。在标准库 `vector` 没有被引入 C++ 标准之前, 很多代码都使用不包含异常处理的 `vector` 类型(非标准), 而在这些代码中, 大部分的代码最终都被修改以使用标准的 `vector` 类型。

19.4.1.2 效率

在极端情况下, 范围检查是一种负担, 例如网络接口的缓冲区、高性能科学计算的矩阵。尽管如此, 在“普通计算”中, 范围检查的代价并不是什么重要话题。因此, 我们建议应该尽量地使用包含范围检查的实现。

19.4.1.3 约束

对于一些程序员和程序而言, 这一观点是成立的。实际上, 这一观点对很多的程序员而言是成立的, 并且不应被忽略。尽管如此, 如果你在一个不涉及硬实时要求(参见 25.2.1 节)的环境中编写程序, 那么你应该选择能够处理异常以及范围检查的 `vector` 类型。

19.4.1.4 检查的可选性

ISO C++ 标准仅仅指出, 它并不保证 `vector` 类型的数据越界访问具有任何特定的语义, 且应尽量避免这类访问。当程序试图进行越界访问时, 抛出异常这一操作实际上是遵循 C++ 标准的操作。因此, 如果你在应用中希望 `vector` 能够抛出异常, 并且不关心前面三个观点, 那么你应该使用包含范围检查的 `vector` 实现。这正是我们在本书中所做的。

概括起来说就是, 现实中的程序设计要比我们所希望的更加复杂混乱, 但总会存在解决问题的办法。

19.4.2 使用宏

与 `vector` 类似, 大多数标准库的 `vector` 类型不对下标操作符(`[]`)进行范围检查, 但在 `at()` 中提供范围检查。那么, 你可能会奇怪, 程序中的 `std::out_of_range` 异常是从何而来呢? 本质上, 我们选择了 19.4.1 节中的“观点 4”: `vector` 的实现并不一定需要对 `[]` 进行范围检查, 但这么做也是可以的, 因此我们实现了这一检查。你之前使用的可能是 `vector` 的调试版本 `Vector`, 而这一版本对 `[]` 进行了范围检查。这一版本是我们开发代码过程中采用的版本。虽然这一版本会牺牲小部分程序性能, 但它有助于减少程序错误和调试时间:

```
struct Range_error : out_of_range { // enhanced vector range error reporting
    int index;
    Range_error(int i) : out_of_range("Range error"), index(i) {}
};

template<class T> struct Vector : public std::vector<T> {
    typedef typename std::vector<T>::size_type size_type;

    Vector() {}
    explicit Vector(size_type n) : std::vector<T>(n) {}
```

```

Vector(size_type n, const T& v) : std::vector<T>(n,v) {}

T& operator[](unsigned int i) // rather than return at(i);
{
    if (i<0||this->size()<=i) throw Range_error(i);
    return std::vector<T>::operator[](i);
}

const T& operator[](unsigned int i) const
{
    if (i<0||this->size()<=i) throw Range_error(i);
    return std::vector<T>::operator[](i);
}
};

```

通过使用 `Range_error`，我们能够对元素的访问索引进行调试。`typedef` 引入了一个更便利的别名，参见 20.5 节。

这个 `Vector` 类十分简单，但它在调试环境中却是十分有用的。另一种方法是使用与测试标准库 `vector` 一样的系统检查——实际上，这可能就是你之前所做的。我们不可能准确地知道你所使用的编译器信息以及你所用的库提供的功能(可能会超出 C++ 标准所要求的功能)。

在 `std_lib_facilities.h` 中，我们采用了一种技巧(宏替代)以重新定义 `vector` 使之代表 `Vector`：
 // disgusting macro hack to get a range-checked vector:
#define vector Vector

这意味着当你使用 `vector` 时，编译器将认为是使用 `Vector`。这一技巧并不太好，因为你所看见的代码与编译器所看见的代码并不相同。在实际的编程中，宏是产生晦涩难懂的错误的一个重要来源(参见 27.8 节和附录 A.17)。

我们同样也为 `string` 类型实现了数据访问的范围检查。

遗憾的是，并不存在标准、简明的方法以对 `vector[]` 的范围检查进行指导。尽管如此，与我们的实现相比，还可以针对 `vector`(和 `string`)实现更简明、更完整的范围检查。然而，这通常涉及更换标准库实现、调整库安装选项或融合标准库的源代码。这些选择对于 C++ 的初学者而言是困难的——我们在第 2 章中就使用了 `string`。

19.5 资源和异常

`vector` 能够抛出异常，并且我们建议，当一个函数不能按要求执行操作时，它应该以抛出异常的方式向其调用者进行报告(参见第 5 章)。现在，是介绍如何处理由 `vector` 操作或者我们调用的其他函数抛出的异常的时候了。一种幼稚的回答是——“使用 `try` 语句块捕获异常，输出一条出错消息并结束程序的运行”——这一方法对于大多数系统而言过于简单了。

编程的一个基本原则是，如果我们获取了资源，那么我们还必须负责(直接或间接地)将其归还给负责管理这些资源的系统。资源的例子包括：

- 内存
- 锁
- 文件句柄
- 线程句柄
- 套接字
- 窗口

本质上，资源可以视为这样的一类东西：资源的使用者必须向系统中的“资源管理者”归还

(释放)资源,并由“资源管理者”负责资源的回收。最简单的例子就是自由空间的内存空间,我们通过 `new` 获得内存空间,而通过 `delete` 归还内存空间。例如:

```
void suspicious(int s, int x)
{
    int* p = new int[s]; // acquire memory
    // ...
    delete[] p;          // release memory
}
```

正如在 17.4.6 节中提到的,我们不得不时刻提醒自己释放内存,但这通常不是那么容易的事情。当我们学习异常处理时,资源泄漏问题变得更为普遍。特别地,我们需要小心处理那些显式使用 `new` 操作并将所得指针赋给局部变量的代码,如 `suspicious()`。

19.5.1 潜在的资源管理问题

我们必须小心处理表面上无害的指针赋值操作,例如

```
int* p = new int[s]; // acquire memory
```

的原因是,在代码中保证每一个 `new` 操作都对应一个 `delete` 操作实际上是很困难的。至少在 `suspicious()` 函数中,必须存在“`delete[] p;`”这样的语句;这样的语句可能会释放内存资源,但也会存在某些意外使得内存的释放不会发生。我们在“...”中放入什么代码才能造成内存泄漏呢?我们的例子应该能为你带来一些启示并引起你对此类代码的警惕。这些例子同时还会使你体会到替代这类代码的简单、有效的方式。

当程序运行到 `delete` 语句时, `p` 可能已经不再指向我们所分配的内存资源:

```
void suspicious(int s, int x)
{
    int* p = new int[s]; // acquire memory
    // ...
    if (x) p = q;        // make p point to another object
    // ...
    delete[] p;          // release memory
}
```

上述例子中的 `if(x)` 使得我们不能确定 `p` 的取值是否已经改变。程序也可能永远都不能到达 `delete` 语句:

```
void suspicious(int s, int x)
{
    int* p = new int[s]; // acquire memory
    // ...
    if (x) return;
    // ...
    delete[] p;          // release memory
}
```

程序不能到达 `delete` 语句的原因也许是程序抛出了一个异常:

```
void suspicious(int s, int x)
{
    int* p = new int[s]; // acquire memory
    vector<int> v;
    // ...
    if (x) p[x] = v.at(x);
    // ...
    delete[] p;          // release memory
}
```

我们这里将着重讨论上面的这个例子。当程序员初次遇见这一问题时,他很可能会认为这是

一个异常处理的问题而不是一个资源管理问题。当得出这一错误的判断后，程序员很可能会通过实现异常捕获以试图解决这一问题：

```
void suspicious(int s, int x) // messy code
{
    int* p = new int[s]; // acquire memory
    vector<int> v;
    // ...
    try {
        if (x) p[x] = v.at(x);
        // ...
    } catch (...) { // catch every exception
        delete[] p; // release memory
        throw; // re-throw the exception
    }
    // ...
    delete[] p; // release memory
}
```

上述解决方法会带来一些额外的代码并造成资源释放代码的重复(`delete [] p;`)，换句话说，这一解决方法是拙劣的；更糟的是，它不能对所有的可能情况进行合理的归纳。下面是一个获取更多资源的例子：

```
void suspicious(vector<int>& v, int s)
{
    int* p = new int[s];
    vector<int> v1;
    // ...
    int* q = new int[s];
    vector<double> v2;
    // ...
    delete[] p;
    delete[] q;
}
```

注意，如果 `new` 操作不能够分配所需内存，它将抛出标准库异常 `bad_alloc`。对于这个例子，`try...catch` 技术也可以用于解决内存泄漏 C++ 问题，但在代码中会包含多个 `try` 语句块，这将造成代码的重复冗余。我们不喜欢重复丑陋的代码，因为“重复”意味着代码的维护代价的增加，而“丑陋”意味着代码难于修改、难于阅读，这同样增加了代码维护的代价。

试一试 在上面的例子中添加 `try` 语句块，以保证在产生异常的所有可能的情况下，资源都能被正确地释放。

19.5.2 资源获取即初始化

幸运的是，我们可以不必在代码中添加复杂的 `try...catch` 语句就能有效处理潜在的资源泄漏问题。例如：

```
void f(vector<int>& v, int s)
{
    vector<int> p(s);
    vector<int> q(s);
    // ...
}
```

这一实现就好得多了。资源（在这里是自由存储区中的内存空间）由构造函数获取，而由对应的析构函数释放。

当我们解决了向量的内存泄漏问题之后，实际上已经解决了这类特别的“异常问题”。这一解

决方法具有通用性，它能用于所有资源类型：通过对象的构造函数获取资源，并通过对应的析构函数释放资源。通过这一方法能够有效处理的资源包括：数据库锁、套接字和 I/O 缓冲区。这一技术有一个拗口的名字“资源获取即初始化”，简称为 RAII。

再回到上面的例子。不论我们采用哪种方式退出函数 $f()$ 、 p 和 q 的析构函数都将被正常调用：因为 p 和 q 不是指针，我们不能对它们赋值，`return` 语句和异常的抛出均不会妨碍析构函数的执行。当程序的执行序列超出了被完全构造的对象或子对象的作用域时，这些对象的析构函数将自动被调用。当一个对象的构造函数执行完毕时，才可以认为它被构造成功。探寻这两句话的详细含义是一件让人头疼的事情，但它们的基本含义是对象的构造函数和析构函数会根据实际需要被调用。

特别地，当需要在某一范围内使用可变大小的存储空间时，我们应使用 `vector` 而不是显式使用 `new` 和 `delete`。

19.5.3 保证

当我们不能只在单一的作用域（及其子作用域）内使用 `vector` 对象时，我们应该怎么做呢？例如：

```
vector<int>* make_vec()    // make a filled vector
{
    vector<int>* p = new vector<int>; // we allocate on free store
    // ... fill the vector with data; this may throw an exception ...
    return p;
}
```

这个例子具有普遍意义：我们调用一个函数构造一个复杂的数据结构，并将该结构作为结果返回。问题是，如果在“填充”`vector` 对象时发生了异常，那么 `make_vec()` 将会造成 `vector` 对象所占内存空间的泄漏。一个不相关的问题是，如果该函数成功了，那么我们不得不通过 `delete` 销毁由 `make_vec()` 返回的对象（参见 17.4.6 节）。

我们可以通过 `try` 语句块处理异常的抛出：

```
vector<int>* make_vec()    // make a filled vector
{
    vector<int>* p = new vector<int>; // we allocate on free store
    try {
        // fill the vector with data; this may throw an exception
        return p;
    }
    catch (...) {
        delete p;    // do our local cleanup
        throw;       // re-throw to allow our caller to deal with the fact
                     // that some_function() couldn't do what was
                     // required of it
    }
}
```

`make_vec()` 函数展示了错误处理的一个十分通用的形式：函数总是试图完成它的工作，而如果它不能完成工作，则它应释放所有的局部资源（在这里是自由存储区中分配的 `vector` 对象）并通过抛出异常的方式报告其工作的失败。在这里，异常是由一些其他的函数产生并抛出的（`vector::at()`）；`make_vec()` 只是通过 `throw` 直接将该异常重新抛出；这是一种简单而有效的处理错误的方法，并且能够被系统地使用：

- 基本保证：代码 `try...catch` 的目的是保证 `make_vec()` 要么成功，要么在不造成资源泄漏的前提下抛出异常，这通常称为基本保证。如果程序中的某段代码需要能够从异常 `throw` 中

恢复,那么该段代码就需要提供基本保证。所有的标准库代码均提供了基本保证。

- **强保证**: 如果一个函数除了提供基本保证,还具有如下特征:在该函数的任务失败后,所有可观测值(所有不属于该函数的值)的取值仍能与其在该函数被调用前的取值一致,那么称该函数提供强保证。强保证是一种理想情况:函数要么成功完成了所有的任务,要么除了抛出异常之外什么也不做。
- **无抛出保证**: 除非我们进行的操作十分简单以至该操作不会产生任何失败和异常的抛出,否则我们很可能不能实现同时满足基本保证和强保证的代码。幸运的是, C++ 提供的所有内建工具本质上能够提供无抛出保证:它们不会抛出异常。为了避免异常的抛出,我们应该避免使用 `throw`、`new` 以及引用类型的 `dynamic_cast`(参见附录 A.5.7)。

基本保证和强保证对于检验程序的正确性是十分有用的。为了能够根据这些理想情况编写高性能的代码,RAII 是必不可少的。有兴趣的读者可以阅读《The C++ Programming Language》一书的附录 E。

我们应该总是避免执行未定义的操作(通常它们是有害的),例如对 0 进行解引用、以 0 为除数、对数组越界访问。捕获异常并不能保证你不违反这些基本的语言规则。

19.5.4 auto_ptr

在出现异常的情况下, `make_vec()` 遵守了资源管理的基本原则。它提供了基本保证——所有实现良好的函数都应该提供基本保证。该函数还提供了强保证,除非该函数在“向 vector 填充数据”这部分代码中对非局部数据进行了操作。尽管如此, `try...catch` 这部分代码仍然是丑陋的。解决方法是:我们必须使用 RAII;也就是说,我们需要提供一个对象以容纳 `vector<int>` 对象,以使得当异常发生时它能够销毁 `vector` 对象。为实现这一目标,在 `<memory>` 中标准库提供了 `auto_ptr`:

```
vector<int>* make_vec() // make a filled vector
{
    auto_ptr<vector<int>> p(new vector<int>); // allocate on free store
    // fill the vector with data; this may throw an exception
    return p.release(); // return the pointer held by p
}
```

`auto_ptr` 对象是一个能够在函数中存储指针的对象。我们通过 `new` 返回的对象初始化 `auto_ptr` 对象。与指针类似,我们可以对 `auto_ptr` 使用 `->` 和 `*` 操作符(例如 `p->at(2)` 或 `(*p).at(2)`),因此我们可以认为 `auto_ptr` 是一类指针。然而,我们不应该在阅读有关 `auto_ptr` 的文档前对 `auto_ptr` 执行拷贝操作;`auto_ptr` 的语义与我们之前遇见的类型不同。`release()` 操作使 `auto_ptr` 返回普通的指针,以使得我们能够将其作为函数返回值;并且使得 `auto_ptr` 在函数返回时不销毁它指向的对象。我们不应采用其他方式(例如拷贝)使用 `auto_ptr`。`auto_ptr` 的用途是存储指针并保证在指针所指向对象的作用域结束时销毁该对象,`auto_ptr` 的其他用途需要你掌握一些相当专业的知识。`auto_ptr` 是保证诸如 `make_vec()` 之类代码的简单性和高效性的一种特殊的工具。特别地, `auto_ptr` 使得我们能够继续保持对显式 `try` 语句块的警惕:大多数的 `try` 语句块能够被“资源获取即初始化”技术取代。

19.5.5 vector 类的 RAII

像 `auto_ptr` 这样带有一点智能的指针看上去有点特别。如何保证我们已经发现了所有需要保护的指针?如何保证我们已经释放了所有指针指向的所有对象?回到 19.3.6 节中的 `reserve()` 例子:

```

template<class T, class A>
void vector<T,A>::reserve(int newalloc)
{
    if (newalloc<=space) return;           // never decrease allocation
    T* p = alloc.allocate(newalloc);       // allocate new space

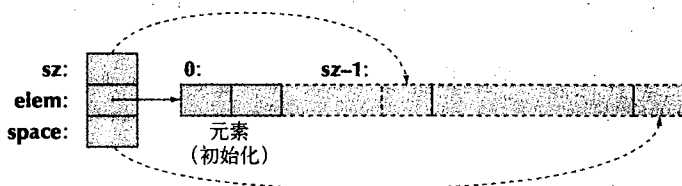
    for (int i=0; i<sz; ++i) alloc.construct(&p[i],elem[i]); // copy

    for (int i=0; i<sz; ++i) alloc.destroy(&elem[i]);         // destroy

    alloc.deallocate(elem,space);           // deallocate old space
    elem = p;
    space = newalloc;
}

```

注意, 对已有元素的拷贝操作 `alloc.construct(&p[i], elem[i])` 可能会抛出异常。因此, `p` 是我们在 19.5.1 节中所描述问题的一个例子。我们可以采用 `auto_ptr` 解决方案。一个更好的解决方案是, 将“向量所占内存”认为是一种资源; 也就是说, 我们可以定义一个 `vector_base` 类以代表我们之前使用过的基本概念。下图显示了向量对象中用于定义内存使用的三个成员:



`vector_base` 的代码(为保持完整性而加入了分配器)的形式为:

```

template<class T, class A>
struct vector_base {
    A alloc;           // allocator
    T* elem;           // start of allocation
    int sz;             // number of elements
    int space;          // amount of allocated space

    vector_base(const A& a, int n)
        : alloc(a), elem(a.allocate(n)), sz(n), space(n) {}
    ~vector_base() { alloc.deallocate(elem,space); }
};

```

注意, `vector_base` 处理的是内存而不是(类型)对象。我们的 `vector` 实现可以将它用于存储所需要元素类型的对象。本质上, `vector` 是 `vector_base` 的一个便捷的接口:

```

template<class T, class A = allocator<T>>
class vector : private vector_base<T,A> {
public:
    // ...
};

```

我们可以按如下方式重新实现 `reserve()`:

```

template<class T, class A>
void vector<T,A>::reserve(int newalloc)
{
    if (newalloc<=this->space) return;           // never decrease allocation
    vector_base<T,A> b(this->alloc,newalloc);    // allocate new space
    for (int i=0; i<this->sz; ++i) this->alloc.construct(&b.elem[i],
        this->elem[i]); // copy
    for (int i=0; i<this->sz; ++i) this->alloc.destroy(&this->elem[i]);
        // destroy old
}

```

```
    swap<vector_base<T,A>>>(*this,b); // swap representations
}
```

当我们退出 `reserve()` 函数时, 原有内存空间将被 `vector_base` 的析构函数自动释放——即使退出是由拷贝操作所产生的异常造成的。`swap()` 函数是一个标准库算法(在 `<algorithm>` 中), 它能够交换两个对象的取值。我们使用 `swap<vector_base<T, A>>>(*this, b)` 而不是 `swap(*this, b)`, 因为 `*this` 和 `b` 是两种不同的类型(`vector` 和 `vector_base`)。类似地, 当我们从派生类 `vector<T, A>` 的成员(如 `vector_base<T, A>::reserve()`)中引用基类 `vector_base<T, A>` 的成员时, 必须显式使用 `this->`。

试一试 通过使用 `auto_ptr` 修改 `reserve` 函数。记住在返回前调用 `release()` 函数。将这种方法与 `vector_base` 方法相比较, 看看哪种方法更容易正确地实现。

简单练习

1. 定义 `template<class T> struct S{T val;};`。
2. 添加构造函数, 使得能够对 `T` 初始化。
3. 定义 `S<int>`、`S<char>`、`S<double>`、`S<string>` 和 `S<vector<int>>` 类型的变量, 并将其初始化。
4. 读取并打印上述变量的值。
5. 添加函数 `get()`, 该函数返回对 `val` 的引用。
6. 在类之外编写 `get()` 的定义。
7. 将 `val` 设为私有成员。
8. 用 `get()` 函数完成练习 4 中的任务。
9. 添加函数模板 `set()` 以能够改变 `val`。
10. 用 `operator[]()` 取代 `get()` 和 `set()`。
11. 编写 `operator[]()` 的 `const` 版本和非 `const` 版本。
12. 定义函数 `template<class T> read_val(T&v)`, 该函数能够将 `cin` 中读取的值写入 `v`。
13. 通过 `read_val()` 设置练习 3 中前 4 个变量的值。
14. 定义 `template<class T> istream &operator>>(istream &, vector<T> &)` 以使 `read_val()` 能够处理 `S<vector<int>>` 变量。

记住在每一步中对代码进行测试。

思考题

1. 为什么需要调整 `vector` 对象的大小?
2. 为什么需要使用具有不同元素类型的 `vector` 对象?
3. 为什么不在所有可能的情况中定义一个具有足够大规模的 `vector` 对象?
4. 需要为一个新的 `vector` 对象分配多少空闲内存空间?
5. 何时必须将 `vector` 对象包含的元素拷贝至新的内存空间?
6. 在一个 `vector` 对象构造成功之后, 哪些 `vector` 操作能够改变它的大小?
7. 拷贝结束后, `vector` 对象的取值如何?
8. 哪两个操作定义了 `vector` 的拷贝?
9. 对于类的对象而言, 拷贝的默认含义是什么?
10. 什么是模板?
11. 最有用的两种模板参数类型是什么?
12. 什么是泛型编程?
13. 泛型编程与面向对象的编程之间有什么区别?
14. `array` 与 `vector` 有什么区别?
15. `array` 与内置数组有什么区别?

16. `resize()` 和 `reserve()` 有什么区别?
17. 什么是资源? 给出它的定义并举例说明。
18. 什么是资源泄漏?
19. 什么是 RAII? 它能解决什么问题?
20. `auto_ptr` 的用途是什么?



术语

<code>#define</code>	宏	特例化	<code>at()</code>
<code>push_back()</code>	强保证	<code>auto_ptr</code>	RAII
模板	基本保证	<code>resize()</code>	模板参数
异常	资源	<code>this</code>	保证
重抛出	<code>throw;</code>	实例化	自我赋值



习题

针对每一习题, 通过已定义类的对象测试你的设计和实现确实达到了预期要求。在涉及异常的地方, 需要认真考虑错误产生的来源。

1. 编写一个模板函数 `f()`, 该函数能够实现将一个 `vector<T>` 中的元素加到另一个 `vector<T>` 中的元素, 例如, `f(v1, v2)` 应对 `v1` 的每个元素计算 `v1[i] += v2[i]`。
2. 编写一个模板函数, 该函数以 `vector<T> vt` 和 `vector<U> vu` 为参数并返回所有 `vt[i] * vu[i]` 之和。
3. 编写一个模板类 `Pair`, 该类能够存储任何类型的值对。使用该实现一个类似于我们在计算器中所使用的符号表(参见 7.8 节)。
4. 将 17.9.3 节中的 `Link` 类实现为模板, 该模板以数值类型作为模板参数。然后使用 `Link<God>` 完成第 17 章的习题 13。
5. 定义 `Int` 类, 该类包含一个 `int` 类的成员。定义该类的构造函数、赋值和 `+`、`-`、`*`、`/` 操作符。测试该类并根据需要对其进行完善(例如, 定义 `<<` 和 `>>` 操作符)。
6. 使用 `Number<T>` 类重新完成上面的习题, 其中 `T` 可以是任何数值类型。为 `Number` 实现 `%` 操作符并观察对 `Number<double>` 和 `Number<int>` 进行 `%` 运算的结果。
7. 通过 `Number` 完成习题 2。
8. 通过基本函数 `malloc()` 和 `free()` (见附录 B10.4) 实现一个分配器(参见 19.3.6 节)。通过见 19.4 节结束时定义的 `vector` 类型测试该分配器。
9. 通过使用分配器(参见 19.3.6 节)重新实现 `vector::operator = ()` (参见 19.2.5 节)。
10. 实现一个简单的 `auto_ptr`, 实现其构造函数、析构函数、`->`、`*` 和 `release()`。特别地, 不要实现其赋值或拷贝构造函数。
11. 设计并实现 `counted_ptr<T>` 类型, 该类型存储两个对象: 一个指向 `T` 类型对象的指针; 一个指向代表该 `T` 对象“使用计数”的整型数的指针, 该整型数被所有指向该 `T` 对象的计数指针(`counted_ptr`)所共享。对于一个确定的 `T` 对象, “使用计数”的取值代表了指向该对象的所有计数指针的总数。`counted_ptr` 的构造函数在自由存储区中为 `T` 对象和其“使用计数”分配内存空间。为 `counted_ptr` 的 `T` 设定一个初始值。当最后一个指向 `T` 对象的 `counted_ptr` 被销毁时, `counted_ptr` 的析构函数应负责销毁 `T` 对象。实现 `counted_ptr` 类型的相关操作, 以使我们能够与使用指针相同的方式使用 `counted_ptr` 类型的对象。这一习题是“智能指针”的一个例子, 它能够保证一个对象的存在, 直至该对象的最后一个使用者停止使用该对象。编写测试 `counted_ptr` 的方案, 并在方案中将 `counted_ptr` 作为调用的参数, 如作为容器的元素, 等等。
12. 定义 `File_handle` 类, 该类的构造函数以一个字符串(文件名)作为参数, 并且该类在构造函数中打开文件, 而在析构函数中关闭文件。
13. 编写 `Tracer` 类, 且该类的构造函数和析构函数都会打印一个字符串, 其中待打印字符串以构造函数参数的形式进行传递。通过 `Tracer` 类观察 RAII 管理对象会在代码中的什么位置完成它们的任务(即通过

Tracer 观察局部对象、成员对象、全局对象由 new 分配的对象,等等)。然后,为 Tracer 类添加拷贝构造函数和拷贝赋值函数,并通过 Tracer 对象观察拷贝是在何时进行的。

14. 为第 18 章习题中的“猎杀怪兽”游戏实现 GUI(用户图形界面)接口和图像输出功能。程序从输入框中获得输入信息,并在一个窗口中显示当前玩家已知的山洞部分。
15. 修改上一习题的程序,以使用户能够在他当前所获得的信息和猜测的基础上标识房间,例如“可能有蝙蝠”和“无底陷阱”。
16. 在有些场合中,人们希望一个空的 vector 对象所占用的内存空间尽可能地小。例如,我们可能需要大量使用 `vector < vector < vector < int >>>` 类型,且大部分的向量均是空向量。定义一个 vector,使得 `sizeof (vector < int >) == sizeof (int*)`,即 vector 只包含一个指针,且该指针指向一个由元素、元素数量值、space 指针组成的结构。

附言

模板和异常是十分强大的语言特征。它们为编程技术带来了相当的灵活性——主要通过帮助人们分离关注点的方式,也就是说,一次只处理一个问题。例如,通过使用模板,我们可以在不关注元素类型的情况下设计一个容器,例如 vector。类似地,通过使用异常,我们能够将用于检测和报告错误的代码和处理该错误的代码相分离。本章的第三个主题,改变 vector 的大小,也体现了这一灵活性: `push_back()`、`resize()` 和 `reserve()` 使我们能够将 vector 的实现与 vector 的大小规格相分离。

第 20 章 容器和迭代器

“只做一件事，并把它做好。多个程序协同工作。”

——Doug Mellory

本章和第 21 章将分别介绍 C++ 标准库 (STL) 中的容器和算法部分。STL 是一个用于处理 C++ 程序中的数据可扩展框架。我们首先通过一个简单的例子来说明 STL 的设计理念和基本概念。我们会详细讨论迭代器、链表和 STL 中的容器。STL 通过序列 (sequence) 和迭代器 (iterator) 的概念将容器 (数据) 和算法 (数据处理方法) 关联起来。本章的内容为第 21 章介绍通用和高效的算法奠定了基础。作为示例，本章实现了一个文字处理程序的基本框架。

20.1 存储和处理数据

在处理数据量很大的问题之前，我们先来看一个简单的例子，它说明了解决一般数据处理问题的基本方法。Jack 和 Jill 分别负责测量过往车辆的速度，结果用浮点数来表示。Jack 是一名 C 语言程序员，所以将测量值保存到一个数组中，而 Jill 将测量值保存到有关 vector 对象中。如果我们要在程序中使用他们的数据，该如何操作呢？

我们可以将 Jack 和 Jill 程序的结果分别写到某个文件中，然后再从文件中读入数据。使用这种方法，我们的程序将与 Jack 和 Jill 所选用的数据结构和接口彻底无关。通常，这种程序之间的独立性是一种很好的特性，此时我们可以采用第 10 和 11 章中介绍的方法来获得输入数据，并利用 `vector<double>` 对象来进行计算。

但是，如果我们的任务不适合使用文件呢？假设，我们必须每秒钟调用一次数据生成函数来获得一组新的数据。例如，下面的程序每秒都会调用 Jack 和 Jill 的函数来获得将要处理的数据：

```
double* get_from_jack(int* count); // Jack puts doubles into an array and
                                   // returns the number of elements in *count
vector<double>* get_from_jill();    // Jill fills the vector

void fct()
{
    int jack_count = 0;
    double* jack_data = get_from_jack(&jack_count);
    vector<double>* jill_data = get_from_jill();
    // ... process ...
    delete[] jack_data;
    delete jill_data;
}
```

上面这段代码假设我们要自己安排存储数据的空间，而且在用完这些数据之后要自己负责删除工作。另一个假设是我们不能重写 Jack 和 Jill 的代码，而且通常我们也不想这样做。

20.1.1 处理数据

显然，这个例子过于简单，但是它与很多实际问题并没有本质区别。如果我们能够很好地解决这个例子，就能够处理一大类通用的编程问题。问题的关键在于，我们无法控制提供数据的程

序以什么形式来提供数据。我们可以自由决定是沿用原有的数据格式，还是转换为另一种形式来进行存储和处理。

我们需要如何处理数据？排序？找出最大值？找出平均值？找出大于 65 的值？比较 Jill 和 Jack 的数据？处理需求多种多样，我们只能根据具体任务来编写处理程序。这里，我们主要是学习怎样处理数据，完成大量数据的计算。首先从简单的处理开始：找到数据集中的最大值。我们可以将 `fcn()` 函数中内容为“...process...”的注释行替换为下面这段代码：

```
// ...
double h = -1;
double* jack_high; // jack_high will point to the element with the highest value
double* jill_high;  // jill_high will point to the element with the highest value

for (int i=0; i<jack_count; ++i)
    if (h<jack_data[i]) {
        jack_high = &jack_data[i]; // save address of largest element
        h = jack_data[i];           //update "largest element"
    }
h = -1;
for (int i=0; i<jill_data->size(); ++i)
    if (h<(*jill_data)[i]){
        jill_high = &(*jill_data)[i]; // save address of largest element
        h = (*jill_data)[i];           //update "largest element"
    }
cout << "Jill's max: " << *jill_high
     << "; Jack's max: " << *jack_high;

// ...
```

注意访问 Jill 的数据时使用的方法：`(*jill_data)[i]`。`get_from_jill()` 函数返回一个指向 `vector` 对象的指针 `vector<double>*`。为了获得数据内容，我们首先要获得 `vector` 对象本身的引用：`*jill_data`，然后通过索引下标访问其中的元素。然而，`*jill_data[i]` 并不是我们想要的结果，因为运算符 `[]` 的优先级要高于运算符 `*`，所以这个表达式的含义是 `*(jill_data[i])`，必须在 `*jill_data` 外使用括号，结果为：`(*jill_data)[i]`。

试一试 如果我们修改 Jill 的代码，应该如何修改代码的接口来避免复杂的数据访问方法？

20.1.2 一般化代码

我们希望使用统一的方法来访问和处理数据，这样可以避免因为每次获得的数据格式不同而编写不同的处理代码。下面我们以 Jack 和 Jill 的代码为例，讨论如何让我们的代码更通用、更统一。

显然，我们对 Jack 和 Jill 的数据的处理方法很相似。但是处理代码有两个不同之处：`jack_count` 和 `jill_data->size()`；`jack_data[i]` 和 `(*jill_data)[i]`。我们可以通过定义下面的引用来避免第 2 个不同之处：

```
vector<double>& v = *jill_data;
for (int i=0; i<v.size(); ++i)
    if (h<v[i]){
        jill_high = &v[i];
        h = v[i];
    }
```

这段代码与处理 Jack 数据的代码很相似。接下来如何编写一个可以同时处理 Jack 和 Jill 数据的函数呢？方法有很多（参考练习 3），出于整体一致性的考虑（这一点在接下来的两章中十分明显），

我们选择下面这种基于指针的方法：

```
double* high(double* first, double* last)
// return a pointer to the element in [first,last) that has the highest value
{
    double h = -1;
    double* high;
    for(double* p = first; p!=last; ++p)
        if (h<*p) {high = p; h = *p;}
    return high;
}
```

使用这个函数，数据处理代码可以改写为：

```
double* jack_high = high(jack_data, jack_data+jack_count);
vector<double>& v = *jill_data;
double* jill_high = high(&v[0], &v[0]+v.size());
```

这段代码更加简洁，不仅省去了很多变量的定义，并且只出现了一段循环代码（在 `high()` 中）。如果我们想要得到最大值，只需要检查 `*jack_high` 和 `*jill_high`，例如：

```
cout << "Jill's max: " << *jill_high
      << "; Jack's max: " << *jack_high;
```

注意，`high()` 函数要求所处理的数据保存在一个数组中，所以“找出最大值”的算法返回的是指向数组元素的指针。

试一试 这段程序中有两个潜在的严重错误。其中一个会导致程序异常，另一个会导致 `high()` 函数返回错误的结果。下面将要介绍的一般化技术会充分暴露出这两个错误，并给出系统的避免方法。现在我们只需要找出这两个错误，并提出修改意见。

`high()` 函数的局限性在于只能处理某个特定的问题：

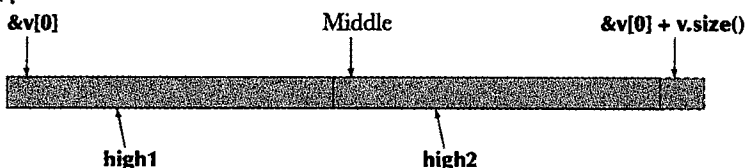
- 只能处理数组。vector 的元素必须保存在数组中，但实际上数据的存储方式还有可能是 list 和 map（参见 20.4 节和 21.6.1 节）。
- 可以处理 double 类型的 vector 或数组，但是无法处理其他类型的元素，例如 `vector<double*>` 和 `char[10]`。
- 只能找出最大值，无法完成其他简单的数据计算功能。

下面，我们探讨如何在更通用的数据集合上进行计算。

通过指针的方式来实现“找出最大值”的算法会带来一个意想不到的通用性：我们不仅可以找出整个数组或 vector 中的最大值，还可以找出数组或 vector 的某个部分的最大值，例如：

```
// ...
vector<double>& v = *jill_data;
double* middle = &v[0]+v.size()/2;
double* high1 = high(&v[0], middle);           // max of first half
double* high2 = high(middle, &v[0]+v.size()); // max of second half
// ...
```

这里 `high1` 指向 vecotr 中前半部分的最大值，`high2` 指向 vecotr 中后半部分的最大值。下面是这个结果的图形表示：



high()函数的参数是指针,这有些偏于底层,更容易引起错误。对于大多数程序员来说,找出 vector 中最大值的代码如下所示:

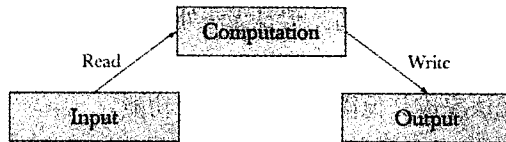
```
double* find_highest(vector<double>& v)
{
    double h = -1;
    double* high = 0;
    for (int i=0; i<v.size(); ++i)
        if (h<v[i]) { high = &v[i]; h = v[i]; }
    return high;
}
```

然而,这段代码就失去了 high()所具有的灵活性:我们不能用 find_highest()来找出 vector 中某一部分的最大值。我们只是为了同时处理数组和 vector 才决定使用指针,但实际上却意外地获得了某种灵活性。我们应该记住:一般化处理可以获得适用于多个问题的通用函数。

20.2 STL 建议

C++ 标准库为处理数据序列提供了一个专门的框架,称为 STL。STL 是标准模板库(standard template library)的简称。STL 是 ISO C++ 标准库的部分,它提供了容器(例如 vector、list 和 map)和通用算法(例如 sort、find 和 accumulate)。因此我们可以称 vector 这类对象为 STL 或标准库的一部分。标准库的其他部分,例如 ostream(见第 10 章)和 C 风格的字符串处理函数(见附录 B.10.3)并不属于 STL。为了更好地理解 STL,我们首先考虑在处理数据时必须解决的问题和对解决方法的基本要求。

计算过程包含两个方面:计算和数据。有时候我们只关注计算方面,谈论 if 语句、循环、函数和错误处理等。有时我们关注的是数据,包括数组、向量、字符串和文件等。然而,要完成真正的工作我们需要同时考虑计算和数据。如果不经分析、可视化和查找所需数据等处理,大量数据都无法使用。相反,我们可以随心所欲地进行计算,但是只有与实际数据关联之后,才能避免没有意义的计算过程。而且,“计算部分”要优雅地与“数据部分”进行交互。



当谈起数据时,我们会想到很多类型的数据:各种形状、数以百计的温度值、数以千计的日志记录、数以百万计的指针和网页等,即数据的容器和数据流。特别地,我们并不是要讨论如何为一种对象(例如,复数、温度值和圆形等)选择最恰当的数值。对于这些数据类型,可以参考第 9、11 和 14 章。

考虑我们需要对“大量数据”进行的一些简单操作:

- 按照字典序排序。
- 根据姓名在电话本中找到对应的电话号码。
- 找出温度的最高值。
- 找出所有大于 8800 的值。
- 找出第一个值为 17 的元素。
- 根据单元编号对遥测记录进行排序。
- 根据时间戳对遥测记录进行排序。

- 找出第一个值大于“Petersen”的元素。
- 找出最大值。
- 找出两个序列的第一个不同之处。
- 计算两个序列的内积。
- 找出一个月中每天的最高气温。
- 在销售记录中找出最畅销的 10 件商品。
- 统计“Stroustrup”在网页中出现的次数。
- 计算各个元素之和。

注意，我们在讨论上述数据处理任务时，并没有提到数据如何存储。很显然，我们必须要在数据处理过程中用到列表、向量、文件和输入流等内容，但是我们不必了解这些数据存储的细节就可以完成相应的处理任务。重要的是这些数据或对象（或元素）的类型、如何访问这些数据或对象以及要对它们进行哪种操作。

现实中存在多种常见的处理任务，这就要求我们编写代码来简单、高效地处理这些任务。对程序员来说，需要考虑的问题有：

- 数据类型多种多样。
- 存在多种可选的数据存储方法。
- 对数据集合的操作也是千变万化。

为了尽可能降低这些问题的影响，我们希望编写的代码能够同时处理各种数据类型，同时处理各种数据存储方法，同时适用于各种处理任务。总之，我们想通过代码一般化来处理各种变化情况。我们要避免对每一个问题都从头开始寻找处理方法，那样会浪费大量的时间。

为了编写能够达到上述目的代码，我们首先用更加抽象的方式来看待我们要对数据进行的处理工作：

- 收集数据并装入容器
 - 例如 vector、list 和数组
- 组织数据
 - 打印
 - 快速访问
- 提取数据
 - 根据索引（例如，第 42 个元素）
 - 根据值（例如，年龄字段是 7 的第一个元素）
 - 根据属性（例如，所有温度字段大于 32 且小于 100 的元素）
- 修改容器
 - 增加数据
 - 减少数据
 - 排序（根据某种规则）
- 进行简单的数值运算（例如，将每一个元素乘以 1.7）

在完成上述任务时要避免陷入各种细节之中：各种容器之间的差别、各个访问数据元素方法的差别和各种数据类型的差别。如果我们能够做到这一点，就等于向编写简单、高效、通用的代码的目标迈出了一大步。

回顾前面几章介绍的编程工具和技术，我们(已经)可以编写功能相似，但与数据类型无关的代码：

- 使用 `int` 与使用 `double` 基本没有差异。
- 使用 `vector<int>` 与使用 `vector<string>` 基本没有差异。
- 使用 `double` 类型的数组与使用 `vector<double>` 基本没有差异。

我们希望只有当我们想要完成一些全新的任务时才需要编写新的代码。而且，我们还希望编写一些能够完成基本任务的代码，以至于即使我们需要使用新的数据存储或处理方法，也无需从头开始编写新的代码。

- 在 `vector` 与数组中查找数据的方法差异不大。
- 查找字符串时不区分大小写与区分大小写的差异不大。
- 使用准确值与使用近似值画图的差异不大。
- 拷贝文件与拷贝 `vector` 对象的差异不大。

根据上面的观察，我们编写的代码应具有以下特点：

- 容易阅读
- 容易修改
- 规范
- 简短
- 快速

为了简化编程工作，我们会：

- 使用统一的方式访问数据
 - 与数据存储方法无关
 - 与数据类型无关
- 使用类型安全的方式访问数据
- 便于遍历数据
- 压缩数据存储空间
- 快速
 - 提取数据
 - 增加数据
 - 删除数据
- 对通用算法提供标准实现
 - 例如拷贝、查找、搜索、排序、求和等

STL 提供了上述功能以及更多的其他功能。STL 不仅是一个强大的功能库，而且是一个兼顾灵活性与实现性能的函数库设计的典范。STL 由 Alex Stepanov 设计，是一个通用、正确和高效的算法框架。其设计理念体现在算法实现的简单性、通用性和优雅性。

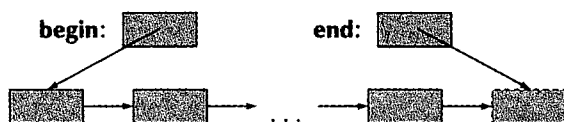
除了使用设计思路清楚、代码逻辑性强的框架来处理数据之外；另一种策略是让程序员根据每个具体任务的特定情况选择处理方法，并且从头开始实现程序代码，这种方式费时费力。不仅如此，得到的程序代码往往非常糟糕，毫无章法可言，除作者之外的人都很难理解，而且在其他场合能够复用的可能性微乎其微。

在介绍完 STL 的设计初衷和理念之后，我们会给出 STL 的一些基本定义，最后通过例子展示

如何在实际应用中运用这些基本理念：编写更好的处理数据的代码，而且让编写过程更简单。

20.3 序列和迭代器

序列是 STL 中的核心概念。从 STL 的角度来看，数据集就是一个序列。序列具有头部和尾部。我们可以对一个序列从头部到尾部进行遍历，对序列中的元素进行有选择的读写操作。我们利用一对迭代器来完成读序列头部和尾部的判断。迭代器是一个可以标识序列中元素的对象。我们可以按照如下图所示的方式来看待一个序列：



这里的 `begin` 与 `end` 就是迭代器，它们标识了序列的头部和尾部。通常称 STL 的序列是“半开”的，因为由 `begin` 所标识的元素是序列的一部分，而迭代器 `end` 通常指向序列尾部后面一个位置。在数学中，这种序列（区间）可以表示为 $[begin: end)$ 。两个元素间的箭头表示如果有一个指向第一个元素的迭代器，那么我们就可以得到一个指向第二个元素的迭代器。

那么究竟什么是迭代器呢？迭代器是一个相当抽象的感念：

- 迭代器指向序列中的某个元素（或者序列末端元素之后）。
- 可以使用 `==` 和 `!=` 来对两个迭代器进行比较。
- 可以使用单目运算符 `*` 来访问迭代器所指向的元素。
- 可以利用操作符 `++` 来使迭代器指向下一个元素。

举例来说，如果 `p` 和 `q` 是两个指向同一个序列的两个迭代器：

标准迭代器的基本操作

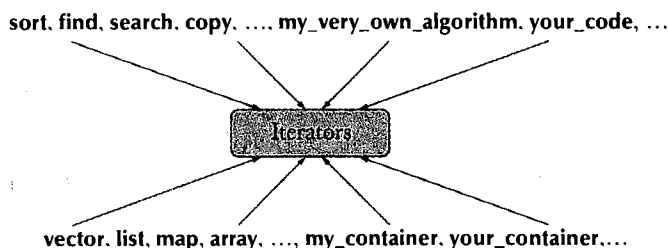
<code>p == q</code>	当且仅当 <code>p</code> 和 <code>q</code> 指向序列中的同一个元素或都指向序列末端元素之后时为真
<code>p != q</code>	<code>!(p == q)</code>
<code>*p</code>	表示 <code>p</code> 所指向的元素
<code>*p = val</code>	对 <code>p</code> 所指向的元素进行写操作
<code>val = *p</code>	对 <code>p</code> 所指向的元素进行读操作
<code>++p</code>	使 <code>p</code> 指向序列中的下一个元素或序列末端元素之后

很明显，迭代器的概念与指针（见 17.4 节）是相关的。实际上，指向数组中某一元素的指针就是一个迭代器。但是，许多迭代器不仅仅是指针；比如说，我们可以定义一个边界检查的迭代器，当试图使它指向 $[begin: end)$ 之外的内容时就会抛出异常。把迭代器定义成一种抽象的标识而不具体指定类型可以给我们带来很大的灵活性和通用性。本章和第 21 章将会举例说明这一点。

试一试 编写一个函数 `void copy(int* f1, int* e1, int* f2)`，该函数把 $[f1: e1)$ 定义的 `int` 型数组复制到数组 $[f2: f2 + (e1 - f1))$ 中。注意只能使用上面所提到的迭代器操作（不能用索引）。

我们可以利用迭代器来实现代码（算法）与数据的连接。程序编写人员了解迭代器的使用方法（并不需要了解迭代器是如何访问数据的），数据提供者向用户提供相应的迭代器而不是数据存储的细节信息。这样得到的是一个简单的程序结构，而且使算法和容器之间保持了很好的独立性。正如 Alex Stepanov 所说：“STL 算法和容器可以共同完成很强大的功能，而这却是因为它们根

本不知道对方的存在。”但是，它们都知道由一对迭代器所定义的序列。



换句话说，我们的代码不再需要知道存储和访问数据的不同方法；它只需要对迭代器有一定的了解。另一方面，作为数据提供方，我们不再需要为不同的用户分别编写代码；我们只需要为数据配备好合适的迭代器。实际上，最简单的迭代器只是由*、++、==、!=等操作所定义的，这无疑会使它变得既方便又快速。

STL 框架包含大概 10 种容器和 60 种由迭代器相连接的算法（参见第 21 章）。另外，许多组织和个人都在开发符合 STL 风格的容器和算法。STL 可能是目前最著名的泛型编程的例子（参见 19.3.2 节）。只要了解了它的基本概念和一些简单的例子，你就可以很容易地掌握其他相关的内容。

20.3.1 回到实例

下面我们来看看如何使用 STL 来描述问题“查找序列中的最大元素”：

```

template<class Iterator>
Iterator high(Iterator first, Iterator last)
// return an iterator to the element in [first:last) that has the highest value
{
    Iterator high = first;
    for (Iterator p = first; p!=last; ++p)
        if (*high<*p) high = p;
    return high;
}
  
```

注意，我们去掉了用来存储当前所找到的最大元素的变量 *h*。当我们不能确定序列中元素的类型时，使用 -1 来完成初始化看起来很奇怪。这是因为它确实很奇怪！而且迟早这会导致一些错误的发生：因为在我们的例子中不会存在负的速度，所以它不会在这里出错。我们要记住像 -1 这样的“魔数”是非常不利于程序的维护的（参见 4.3.1 节、7.6.1 节、10.11.1 节等）。它会限制住我们所定义的函数的使用范围，并说明我们对问题还没有形成一个比较全面的认识，也就是说，“魔数”是一种偷懒的表现。

注意，这里的 *high()* 可以被用于所有可以使用 < 进行比较的元素类型。比如，我们可以利用 *high()* 来查找 `vector<string>` 中按字典顺序最靠后的字符串（参见练习 7）。

high() 模板函数可以在任何由一对迭代器定义的序列中使用。举例来说，可以按照如下方式改写我们的例子：

```

double* get_from_jack(int* count); // Jack puts doubles into an array and
                                   // returns the number of elements in *count
vector<double>* get_from_jill();   // Jill fills the vector

void fct()
{
    int jack_count = 0;
    double* jack_data = get_from_jack(&jack_count);
  
```

```

vector<double>* jill_data = get_from_jill();

double* jack_high = high(jack_data, jack_data+jack_count);
vector<double>& v = *jill_data;
double* jill_high = high(&v[0], &v[0]+v.size());
cout << "Jill's high " << *jill_high << "; Jack's high " << *jack_high;
// ...
delete[] jack_data;
delete jill_data;
}

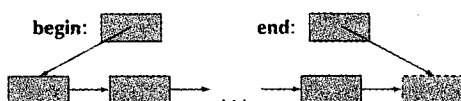
```

这里的两个调用中，`high()` 中的迭代器模板参数的类型为 `double*`。除了确保 `high()` 被正确实现外，这和我们之前的例子没有任何区别。更具体地说，所运行的代码并没有什么不同，但在代码的通用性上却有很大的区别。这里的模板版本的 `high()` 适用于任何由一对迭代器所定义的序列。在进一步了解 STL 的细节和所提供的算法之前，我们先来了解集中对数据元素集合进行存储的方法。

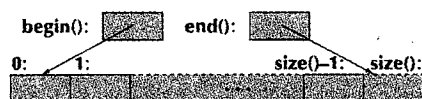
试一试 在我们的程序中有一个严重的错误。请找到并修改它，并提出一种针对这种问题的通用解决方法。

20.4 链表

下面让我们再回顾一下序列概念的图形表示：



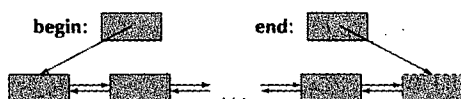
将它与我们描绘 `vector` 内存结构的示意图相比较：



下标 0 本质上与迭代器 `v.begin()` 一样都指向同一个元素，并且下标 `v.size()` 与 `v.end()` 一样都指向最后一个元素之后的位置。

`vector` 的元素在内存中是连续排列的。而在 STL 的概念中，序列在内存中的排列不一定是连续的，因此在 STL 中，很多算法在将一个元素插入已有元素的中间时都不需要移动已有的元素。上面对序列抽象概念的图形描述意味着，在不移动其他元素的前提下进行元素插入（或元素删除）操作成为了可能。STL 迭代器的概念支持了上述操作。

体现上述 STL 示意图所描绘序列概念的最常见的数据结构是链表。在抽象模型中的箭头通常由指针实现。链表中的一个元素是“链接”的一部分，该“链接”由这一元素以及一个或多个指针组成。如果链表的一个链接只包含一个指针，我们称这样的链表为单向链表，如果一个链接包含一个指向前驱链接的指针以及一个指向后继链接的指针，则这样的链表为双向链表。在后续小节中，我们将勾画一个双向链表的实现，且该实现与 C++ 标准库 `list` 的实现相同。双向链表的概念可以由图形描绘，如下所示：

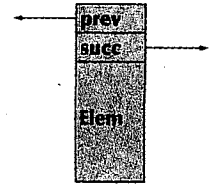


上述概念可由下列代码实现：

```
template<class Elem> struct Link {
    Link* prev; // previous link
    Link* succ; // successor (next) link
    Elem val;   // the value
};

template<class Elem> struct list {
    Link<Elem>* first;
    Link<Elem>* last; // one beyond the last link
};
```

Link 的结构如右图所示。链表的实现和表示方法有多种，附录 B 中给出了标准库所采用的一种方法。在本节中，只概述一下链表的关键属性——我们能够在不影响其他已有元素的前提下插入和删除元素——展示如何在链表中进行迭代，以及给出链表使用方法的一个示例。



当你考虑使用链表时，我们强烈建议你将自己所考虑的链表操作通过图形进行描述。图形是描绘链表操作的一种十分有效的方法。

20.4.1 列表操作

对于列表，我们需要使用哪些操作呢？

- 对 vector 所实现的操作(构造函数、大小等)，除了下标操作。
- 插入(添加一个元素)和删除(移除一个元素)。
- 访问元素以及遍历列表：迭代器。

在 STL 中，上述迭代器的类型是 list 类的一个成员，因此有：

```
template<class Elem> class list {
    // representation and implementation details
public:
    class iterator; // member type: iterator

    iterator begin(); // iterator to first element
    iterator end(); // iterator to one beyond last element

    iterator insert(iterator p, const Elem& v); // insert v into list after p
    iterator erase(iterator p); // remove p from the list

    void push_back(const Elem& v); // insert v at end
    void push_front(const Elem& v); // insert v at front
    void pop_front(); // remove the first element
    void pop_back(); // remove the last element

    Elem& front(); // the first element
    Elem& back(); // the last element

    // ...
};
```

就像“我们的”vector 并没有完全实现标准库 vector 一样，上述 list 定义与标准库 list 定义也不完全相同。但上述 list 中没有任何错误；它仅仅是不完全而已。“我们的”list 的目的在于加深你对链表的理解：链表是什么，list 应该如何实现，以及如何使用 list 的关键特性。如果读者想获得更多的信息，请参考附录 B 或其他专家级别的 C++ 书籍。

迭代器是 STL list 定义中的核心部分。迭代器用于标示元素插入的位置以及待删除的元素(擦除)。它们也可用于在链表中进行“导航”。迭代器的这一用途与我们在 20.1 节和 20.3.1 节

中使用指针遍历数组和向量十分相似。迭代器的这一风格对于标准库算法而言十分关键(参见 21.1 ~ 21.3 节)。

为什么不在 list 中使用下标操作呢? 我们可以为 list 实现下标操作, 但下标操作是一种极为缓慢的操作: list[1000] 操作将会从第一个元素开始访问, 直到被访问元素的数目到达 1000 为止。如果想要这么做, 那么我们可以自己实现这一操作(或使用 advance(); 参见 20.6.2 节)。因此, 标准库 list 并没有提供下标语法。

将迭代器的类型作为 list 的成员(一个嵌套类)的原因在于, 我们没有任何理由将迭代器的类型实现为全局类。这一迭代器的类型将只会由 list 类使用。另外, 这也使得我们能够将每个容器的迭代器都命名为 iterator。在标准库中存在着 list<T>::iterator、vector<T>::iterator、map<K, V>::iterator 等迭代器类型。

20.4.2 迭代

列表迭代器必须提供*、++、== 和!= 操作。因为标准库中的列表为双向链表, 它还提供了-- 操作, 以实现链表的“从后”往前的迭代操作:

```
template<class Elem> class list<Elem>::iterator {
    Link<Elem>* curr;    // current link
public:
    iterator(Link* p) : curr(p) {}

    iterator& operator++() { curr = curr->succ; return *this; } // forward
    iterator& operator--() { curr = curr->prev; return *this; } // backward
    Elem& operator*() { return curr->val; } // get value (dereference)

    bool operator==(const iterator& b) const { return curr==b.curr; }
    bool operator!=(const iterator& b) const { return curr!=b.curr; }
};
```

这些函数十分简明且极具效率: 函数实现中不存在循环, 不存在复杂的表达式, 不存在“可疑的”函数调用。如果你还不清楚这些实现的意义, 请再快速回顾一下前面的示意图。这一 list 迭代器只是一个指向链接的指针。注意, 尽管 list<Elem>::iterator 的实现(代码)与我们在 vector 和数组中用作迭代器的简单指针的实现极为不同, 但两者操作的意义(语义)是相同的。List iterator 提供了对 Link 指针的++、--、*、== 和!= 操作。

现在让我们再次回顾 high() 的实现:

```
template<class Iterator>
Iterator high(Iterator first, Iterator last)
// return an iterator to the element in [first,last) that has the highest value
{
    Iterator high = first;
    for (Iterator p = first; p!=last; ++p)
        if (*high<*p) high = p;
    return high;
}
```

我们可以将其用于 list:

```
void f()
{
    list<int> lst;
    int x;
    while (cin >> x) lst.push_front(x);

    list<int>::iterator p = high(lst.begin(), lst.end());
```

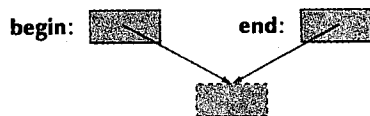
```
    cout << "the highest value was " << *p << endl;
}
```

在上述代码中, Iterator 参数的“取值”为 `list <int> :: iterator`, 并且 `++`、`*` 和 `!=` 操作的实现都与在数组情况下有很大不同, 但操作意义是相同的。模板函数 `high()` 仍然遍历数据 (在这里是 `list`) 和查找最大值。我们可以在 `list` 的任何位置插入一个元素, 因此使用了 `push_front()` 在链表首部添加元素, 而这一操作的目的是为了显示我们确实能够这么做。当然, 也可以像对 `vector` 使用 `push_back()` 函数一样对 `list` 使用 `push_back()` 函数。

试一试 标准库 `vector` 不提供 `push_front()`。为什么? 为 `vector` 实现 `push_front()` 并将其与 `push_back()` 进行比较。

现在, 是时候提出这样的问题了, “如果 `list` 为空会怎样?” 换句话说, “如果 `lst.begin() == lst.end()` 会怎样?” 在这种情况下, `*p` 将会试图对最后一个元素之后的位置进行解引用, `ls.end()`: 这是一个灾难! 或者 (可能更糟地) 结果可能是一个错误的随机值。

上面这个问题的形成给我们带来了一个提示: 我们可以通过比较 `begin()` 和 `end()` 测试一个链表是否为空——实际上, 我们可以通过比较序列的开始和结束判断任何 STL 序列是否为空, 如右图所示。这是使序列的 `end` 指向最后一个元素之后的位置而不是指向最后一个元素的一个更深层次的原因: 空序列不是一种特殊情况。我们不喜欢特殊情况, 因为 (根据定义) 我们不得不为这些特殊的情况编写特殊的代码。



在我们的例子中, 我们可以按如下方式对 `list` 进行测试:

```
list<int>::iterator p = high(lst.begin(), lst.end());
if (p==lst.end())    // did we reach the end?
    cout << "The list is empty";
else
    cout << "the highest value is " << *p << endl;
```

我们采用这种形式的测试方法系统地测试 STL 算法。

因为标准库提供了链表, 我们在这里不再继续深入探讨它的具体实现。取而代之的是, 我们将简要讨论链表适用的场合 (参考练习 12 ~ 14, 如果你对链表的实现细节感兴趣)。

20.5 再次一般化 vector

显然, 通过 20.3 ~ 20.4 节的例子我们发现, 标准库向量包含一个 `iterator` 成员类型以及 `begin()` 和 `end()` 成员函数 (与 `std::list` 类似)。然而, 我们并没有在第 19 章中为我们的 `vector` 类提供这些成员。那么, 对于不同类型的容器而言, 它们究竟采用了什么方法, 以使它们或多或少地能够在 20.3 节所表达的 STL 泛型编程风格中被互换地使用? 首先, 我们将简要介绍一种解决方案 (为简化方案, 我们忽略了分配器), 然后再对解决方案进行解释:

```
template<class T> class vector {
public:
    typedef unsigned long size_type;
    typedef T value_type;
    typedef T* iterator;
    typedef const T* const_iterator;

    // ...

    iterator begin();
    const_iterator begin() const;
    iterator end();
```

```
const_iterator end() const;
```

```
size_type size();
```

```
// ...
```

```
};
```

typedef 为一个类型提供了别名，也就是说，对于我们的 vector，iterator 是我们在 vector 中用做迭代器的类型——T* 的一个同义字，一个别名。现在，对于 vector 对象 v，我们可以编写如下代码：

```
vector<int>::iterator p = find(v.begin(), v.end(), 32);
```

以及

```
for (vector<int>::size_type i = 0; i < v.size(); ++i) cout << v[i] << '\n';
```

通过别名的方式，我们事实上不需要知道 iterator 和 size_type 的实际类型。特别地，因为上述代码使用了 iterator 和 size_type，也可以用于那些 size_type 不为 unsigned long 类型（在很多嵌入式系统中，size_type 为其他类型）并且 iterator 为类而不是简单指针（这种情况在 C++ 实现中很普遍）的 vector 对象。

标准以相似的方式定义了 list 和其他标准容器。例如：

```
template<class Elem> class list {
public:
    class Link;
    typedef unsigned long size_type;
    typedef Elem value_type;
    class iterator;    // see §20.4.2
    class const_iterator; // like iterator,
                        // but not allowing writes to elements

    // ...

    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;

    size_type size();

    // ...
};
```

因此，我们可以编写自己的代码，而不需要关心代码使用了 list 还是向量。所有标准库算法中都使用了上述这些容器的成员类型名字，例如 iterator 和 size_type，所以，算法的实现不依赖于容器的实现或者它们具体操作的容器（参见第 21 章）。

20.6 实例：一个简单的文本编辑器

列表最重要的性质就是可以在不移动元素的情况下对其进行插入或删除操作。下面我们通过一个例子来说明这一点。考虑应该如何在文本编辑器中表示一个文本文件中的字符。所选用的表示方式应当能够使对文本文件进行的操作简单而且高效。

那么具体会涉及哪些操作呢？假设文件存储在计算机的内存中。也就是说，我们可以选择任何一种表示方式，而当需要保存到文件中时，只要把它转换成一个字节流就可以了。相似地，我们也可以把一个文件中的字符转成字节流，从而把它读入到内存中。这说明我们只需要

选择一种合适的内存中的表示方式就可以了。所选择的表示方式需要能够很好地支持以下 5 种操作：

- 从输入的字节流创建该表示方式。
- 插入一个或多个字符。
- 删除一个或多个字符。
- 查找一个 string。
- 产生一个字节流从而输出到文件或屏幕。

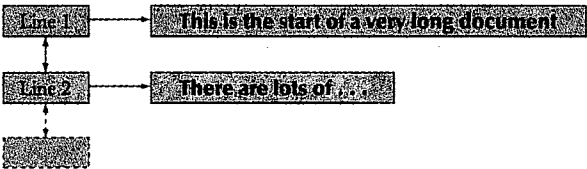
最简单的表示方式就是 `vector<char>`。但是，在 `vector` 中，每加入或删除一个元素时我们就需要移动后面所有的元素。例如：

This is he start of a very long document.
There are lots of ...

我们希望加入字符 `t`：

This is the start of a very long document.
There are lots of ...

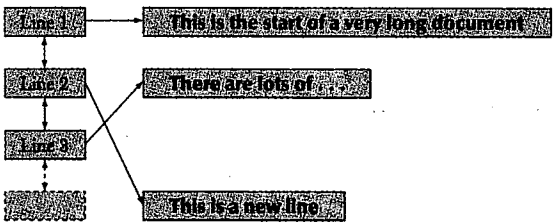
如果用 `vector<char>` 来存储这些字符，那么就需要把从 `h` 开始的所有字符都向右移动，这可能会导致大量的复制操作。实际上，如果要在 70 000 字的文本中插入或删除一个字符，那么平均需要移动 35 000 个字符。它所需要的执行时间会很长，使我们难以接受。因此，不妨把我们的表示方式分成几块，这样就可以在不需移动很多元素的情况下对文本的某一部分进行修改。我们把文本文件看成由一系列“行”组成，并用 `list<Line>` 进行表示，其中 `Line` 是一个 `vector<char>`。例如：



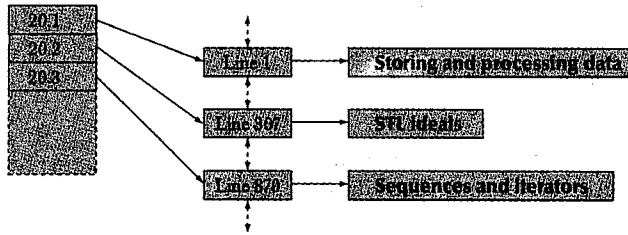
现在，当我们需要插入字符 `t` 时，只需要移动相应一行中的元素就可以了。而且，如果需要的话，我们可以插入新的一行而不需要移动任何元素。例如，我们可以在“document”后面加入字符串“This is a new line.”，可以得到

This is the start of a very long document.
This is a new line.
There are lots of ...

而我们需要做的只是在它们中间加入一行：



我们可以在不影响现有连接的情况下在 `list` 中加入新的连接。这一点非常重要，因为相应的迭代器正是指向现有连接的。这样迭代器就不会被对 `list` 的插入或删除操作影响。比如，文字处理器利用 `vector<list<Line>::iterator>` 来存储指向当前 Document 的标题和子标题的迭代器：



我们可以在不影响指向“20.3 段”的迭代器的情况下向“20.2 段”加入新的行。

也就是说，出于性能和逻辑上的考虑，我们采用的是一个针对“行”的 list，而不是“行”的 vector 或一个存储所有字符的 vector。需要注意的是，由于这种情况很少出现，我们前面提到的“尽量使用 vector”的准则仍然适用。如果要用 list 替代 vector，那么你最好有充分的理由说服你自己（参见 20.7 节）。无论 list 还是 vector 都可以表示逻辑上的“列表”结构。STL 中和我们日常所说的列表最为详尽的是序列，而绝大多数序列是由 vector 表示的。

20.6.1 处理行

应该如何在文件中判断一行呢？有三种方法很容易想到：

- 1) 靠换行符（例如 '\n'）来判断。
- 2) 通过某种自然语言处理的方式，对文件进行分析，从而进行判断（如 .）。
- 3) 把所有过长的行（比如超过 50 个字符）都分成两行。

当然还有其他方法。这里我们选择第一种方法来判断。

我们把编辑器中的文件表示成 Document 类的一个对象，如下所示：

```
typedef vector<char> Line; // a line is a vector of characters
```

```
struct Document {
    list<Line> line; // a document is a list of lines
    // line[i] is the ith line
    Document() { line.push_back(Line()); }
};
```

每个 Document 对象都以一个空行开始：Document 的构造函数会创建一个空行并把它加入到行的列表中。

把文件分行的操作可以按照如下方式完成：

```
istream& operator>>(istream& is, Document& d)
{
    char ch;
    while (is.get(ch)) {
        d.line.back().push_back(ch); // add the character
        if (ch=='\n')
            d.line.push_back(Line()); // add another line
    }
    return is;
}
```

vector 和 list 都有一个可以返回对末端元素引用的 back() 操作。但使用 back() 时一定要确定确实存在末端元素：不要在一个空容器上使用它。这也是为什么我们会对每一个 Document 对象插入一个空 Line 的原因。注意我们会对输入的每个字符都进行存储，包括换行符（'\n'）。这会给我们后面的判断带来很大的方便。

20.6.2 迭代

如果用 vector<char> 来存储文件，那么在它上面使用迭代器就方便多了。那么如何在一个行的列表上使用迭代器呢？我们当然可以使用 list<Line>::iterator。但如果我们希望可以在不考虑

断行的同时对字符进行处理呢？为此，我们针对 Document 类定义一个专门的迭代器：

```
class Text_iterator { // keep track of line and character position within a line
    list<Line>::iterator ln;
    Line::iterator pos;
public:
    // start the iterator at line ll's character position pp:
    Text_iterator(list<Line>::iterator ll, Line::iterator pp)
        :ln(ll), pos(pp) {}

    char& operator*() { return *pos; }
    Text_iterator& operator++();

    bool operator==(const Text_iterator& other) const
        { return ln==other.ln && pos==other.pos; }
    bool operator!=(const Text_iterator& other) const
        { return !(*this==other); }
};

Text_iterator& Text_iterator::operator++()
{
    if (pos==(ln.end())) {
        ++ln; // proceed to next line
        pos = (ln.begin());
    }
    ++pos; // proceed to next character
    return *this;
}
```

为了更好地发挥 Text_iterator 的作用，我们为 Document 定义 begin() 和 end() 操作：

```
struct Document {
    list<Line> line;

    Text_iterator begin() // first character of first line
        { return Text_iterator(line.begin(), (*line.begin()).begin()); }
    Text_iterator end() // one beyond the last line
    {
        list<Line>::iterator last = line.end();
        --last; // we know that the document is not empty
        return Text_iterator(last, (*last).end());
    }
};
```

上面的 (*line.begin()).begin() 是为了使我们可以访问 line.begin() 所指向的数据。当然，由于标准库的迭代器指针 -> 操作符，我们也可以使用 line.begin() -> begin()。

现在我们可以按照如下方式对文件的字符进行访问了：

```
void print(Document& d)
{
    for (Text_iterator p = d.begin(); p!=d.end(); ++p) cout << *p;
}

print(my_doc);
```

把文件作为一个字符序列来处理可以给我们带来很大的方便，但有时候我们所需要的处理对象并不是文件中的某个字符。例如，下面的代码可以删除 Document 中的第 n 行：

```
void erase_line(Document& d, int n)
{
    if (n<0 || d.line.size()-1<n) return; // ignore out-of-range lines
    d.line.erase(advance(d.line.begin(), n));
}
```

函数 `advance(n)` 使迭代器向前移动 n 个元素；`advance()` 是标准库中的函数，不过我们也可以自己对它进行定义：

```
template<class Iter> Iter advance(Iter p, int n)
{
    while (n>0) { ++p; --n; }    // go forward
    return p;
}
```

注意，可以用 `advance()` 来模拟下标机制。实际上，给定一个 `vector v`，那么 `*advance(v.begin(), n)` 和 `v[n]` 大体上是一样的。之所以说是“大体一样”，是因为 `advance()` 会使迭代器一个元素一个元素地向前移动，直到经过第 $n-1$ 个元素位置为止；而 `v[n]` 则会直接把迭代器指向第 n 个元素的位置。

如果迭代器对向前移动和向后移动都支持，比如 `list` 的迭代器，那么为 `advance()` 函数传递负的参数就会使迭代器向后移动。而对支持索引的迭代器，标准库中的 `advance()` 函数会直接移动到指定的位置，而不会用 `++` 操作符一步一步地移动。看来标准库中的 `advance()` 函数确实比我们自己定义的要高级一些。这点很值得注意：一般情况下，标准库中定义好的操作会更仔细地对性能进行考虑，所以还是优先使用它们吧。

试一试 重新实现 `advance()` 函数，使它当输入负的参数时可以向后移动。

对用户来讲，查找可能是最直观的一种迭代了。我们会查找某一个单词（例如 `milkshake` 或 `Gavin`），某一个不能被看做词组的字符序列（例如 `secret\nhomestead`，即前一行以 `secret` 结尾，而后一行以 `homestead` 开始）或一些表达式（例如 `[bB]\w*ne`，即表示以大写或小写字母 `B` 开头，后接 0 个或多个字母，以 `ne` 结尾的串，详见第 23 章）等。下面我们看看如何来处理第二种情况：在 `Document` 中查找某一给定的字符串。注意，我们所采用的算法很简单，但不是最优的：

- 在文件中查找字符串的第一个字符。
- 判断该字符以及其后的字符是不是我们所需要的字符串。
- 如果是，则结束；否则，继续查找字符串的第一个字符。

我们采用 STL 的方式把要被查找的文件当做由一对迭代器定义的字符序列来处理。这样我们既可以对文件的一部分进行查找，也可以对整个文件进行查找。如果在文件中找到了所要的字符串，那么我们返回一个指向该字符串第一个字符的迭代器；否则返回一个指向序列末端的迭代器：

```
Text_iterator find_txt(Text_iterator first, Text_iterator last, const string& s)
{
    if (s.size()==0) return last; // can't find an empty string
    char first_char = s[0];
    while (true) {
        Text_iterator p = find(first,last,first_char);
        if (p==last || match(p,last,s)) return p;
    }
}
```

STL 中一般用返回序列的末端来表示“没有找到”。`match()` 函数非常简单，它只是对两个字符序列进行比较。你可以试着自己来定义它，用来在字符序列中查找某一给定字符的 `find()` 函数可能是标准库中最简单的算法了（参见 21.2 节）。可以按照如下方式使用 `find_txt()` 函数：

```
Text_iterator p =
    find_txt(my_doc.begin(), my_doc.end(),"secret\nhomestead");
if (p==my_doc.end())
    cout << "not found";
else {
    // do something
}
```

我们的文本处理器非常简单。很明显，我们注重的是它的简单性和高效性，而不是功能的复杂性。但不要认为提供有效的插入、删除或查找操作是一件非常简单的事情。我们通过这个例子说明了 STL 序列的强大和通用性，还进一步介绍了迭代器、容器（比如 list 和 vector）以及一些 STL 的通常做法（比如用返回序列的末端来表示操作的失败）。如果需要，我们也可以把 Document 定义成一个 STL 容器——实际上完成对 Text_iterator 的定义也就完成了其中的一大部分工作了。

20.7 vector、list 和 string

为什么对行用 list 而对字符用 vector 呢？更进一步讲，为什么要用 list 处理行的序列而用 vector 处理字符序列呢？再有，为什么不用 string 来存储一行呢？

我们可以把问题再一般化一下。到现在为止，我们知道了 4 种存储字符序列的方法：

- char[] (字符数组)
- vector<char>
- string
- list<char>

那么给定一个具体问题后，我们应该如何选择所采用的存储方式呢？当问题比较简单时，选择哪种方式都无所谓，因为它们都有非常相似的接口。比如，给定一个 iterator，我们可以利用 ++ 和 * 操作符来对字符进行访问。在与 Document 相关的例子中，我们可以很容易地把 vector<char> 换成 list<char> 或 string。这使我们可以根据对性能的要求来选择具体的存储方式。但是，在考虑性能之前，我们先来看看每种存储方式的逻辑特性：能做什么和不能做什么？

- Elem[]: 不知道它自己的大小。没有 begin()、end() 这样的容器成员函数，不能系统地实现边界检查。可以作为参数传递给用 C 或 C 风格的函数。其中的元素在内存中被连续地存储。数组的大小在编译时就确定了。比较(== 和 !=) 和输出(<<) 操作使用的是指向数组第一个元素的指针。
- vector<Elem>: 基本上可以做所有事，包括 insert() 和 erase()。支持索引。支持像 insert() 和 erase() 这样需要移动字符的列表操作(当元素的大小或数目很大时效率会比较低)。支持边界检查。元素在内存中连续存储。vector 可以扩展(例如使用 push_back())。向量的元素被存储为数组。支持对元素进行比较的比较操作符(==、!=、<、<=、> 和 >=)。
- string: 提供了所有常用的操作和文本处理操作，例如字符串的连接(+ 和 +=)。其中的元素不一定在内存中被连续存储。string 可以扩展。支持对元素进行比较的比较操作符(==、!=、<、<=、> 和 >=)。
- list<Elem>: 提供了除索引外所有常用的操作。我们可以在不移动其他元素的情况下 insert() 或 delete() 元素。每个元素需要两个额外的字(指针)来存储。list 可以扩展支持对元素进行比较的比较操作符(==、!=、<、<=、> 和 >=)。

正如我们之前提到的(见 17.2、18.5 节)，当我们需要和底层内存打交道或需要和 C 程序打交道时数组是非常有用且必需的(见 27.1.2 节、27.5 节)。在其他情况下，由于 vector 更方便灵活，常常是我们的首选。

试一试 上述的区别在实际的代码中意味着什么？分别定义一个存有数据“Hello”的 char、vector<char>、list<char> 和 string，并把它们作为参数传递给一个函数。该函

数首先输出其中的数据，并和字符串“Hello”相比较（来判断你是否真的在它们之中存储了“Hello”），然后再和字符串“Howdy”比较，看看它们在字典中谁更靠前。把参数复制到另一个相同类型的变量中。

试一试 重复上面的“试一试”内容，只不过这次不考虑 string，并把所存的数据定义为 {1, 2, 3, 4, 5}。

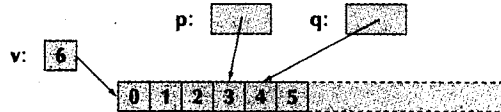
20.7.1 insert 和 erase

标准库中的 vector 是我们使用容器时的首选。它几乎支持所有相关的操作，所以我们只有在没有办法时才会使用其他的替代品。vector 主要的问题在于每当我们执行 insert() 或 erase() 这样的列表操作时，它都需要对元素进行移动。当 vector 很大或其中所存储的元素很大时这会大大降低代码的性能。但也不用太担心这一点。我们可以放心地用 push_back() 来对 500 000 个浮点型数据进行读取。毕竟对性能做一个比较精确的估计是一件不太容易的事。

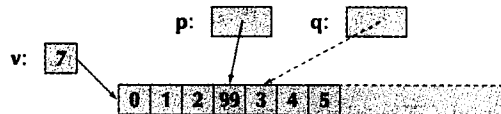
正如在 20.6 节中所提到的，在执行 insert()、erase()、push_back() 等列表操作时，一定不要使迭代器或指针指向 vector 的元素，因为由于元素的移动，这会使迭代器或指针指向错误的位置，这也正是 list 优于 vector 的地方。如果需要在程序中指向许多数量很多而又很大的对象，应该考虑使用 list。

我们来比较一下 list 和 vector 的 insert() 和 erase() 操作。下面的例子可以很好地说明问题：

```
vector<int>::iterator p = v.begin(); // take a vector
++p; ++p; ++p;                      // point to its 4th element
vector<int>::iterator q = p;
++q;                                 // point to its 5th element
```

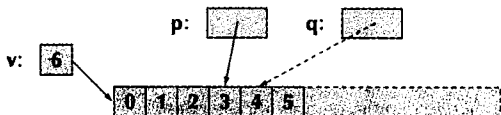


```
p = v.insert(p, 99); // p points at the inserted element
```



现在 q 是无效的。vector 中的元素随着 vector 大小的增加会被重新配置。如果 vector 没有重新分配空间的话，q 应该指向的是值为 3 的元素而不是值为 4 的元素，但千万不要认为一定会是这样。

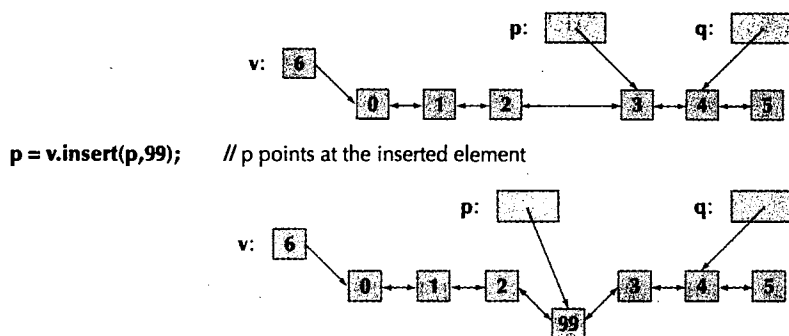
```
p = v.erase(p); // p points at the element after the erased one
```



也就是说，如果在 insert() 操作后再执行一次 erase() 操作，那么 vector 的情况和最开始是一样的，只不过 q 变成无效的了。但是，在这两次操作之间，可能 vector 中所有的元素都被重新分配空间了。

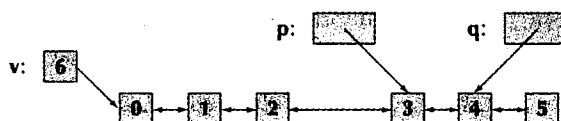
作为对比，我们使用 list 来完成相同的操作：

```
list<int>::iterator p = v.begin(); // take a list
++p; ++p; ++p;                   // point to its 4th element
list<int>::iterator q = p;
++q;                             // point to its 5th element
```



注意, q 仍然指向取值为 4 的元素。

p = v.erase(p); // p points at the element after the erased one



同样, 我们又一次回到了一开始的情况。但是, 与 vector 的不同之处在于, 我们不会移动任何元素, 而且 q 始终是有有效的。

list < char > 与其他 3 种容器相比需要至少 3 倍的存储空间——在 PC 上, 一个 list < char > 需要 12 个字节, 而 vector < char > 只需要 1 个字节。当字符数很多时, 这个因素必须要考虑。

那什么时候能体现出 vector 比 string 好的地方呢? 从它们的性质中可以发现, string 可以完成比 vector 更多的功能。这也正是问题的所在: 由于 string 可以完成的功能太多, 对它进行优化就变得很困难了。实际上, vector 对像 push_back() 这样的内存操作确实是进行了优化的, 而 string 没有。取而代之的是, string 对复制操作、对处理短的字符串的操作以及对与 C 风格字符串的交互进行了优化。在文本编辑器的例子中, 我们选择 vector 是因为我们需要使用 insert() 和 delete(), 这也是出于性能的考虑。vector 和 string 在逻辑上最主要的区别在于 vector 可以用于任何类型的元素, 而只有在处理字符类型的元素时我们才会考虑 string。简而言之, 只有当需要进行字符串操作 (例如字符串连接等) 时才考虑使用 string, 其他情况下, 就用 vector 好了。

20.8 调整 vector 类达到 STL 版本的功能

我们在 20.5 节中为 vector 加入了 begin()、end() 和 typedef 操作, 现在只需要再加入 insert() 和 erase() 操作就可以使我们的 vector 基本符合 std::vector 的要求了:

```
template<class T, class A = allocator<T>> class vector {
    int sz;           // the size
    T* elem;          // a pointer to the elements
    int space;        // number of elements plus number of free space "slots"
    A alloc;          // use allocate to handle memory for elements
public:
    // ... all the other stuff from Chapter 19 and §20.5 ...
    typedef T* iterator; // Elem* is the simplest possible iterator

    iterator insert(iterator p, const T& val);
    iterator erase(iterator p);
};
```

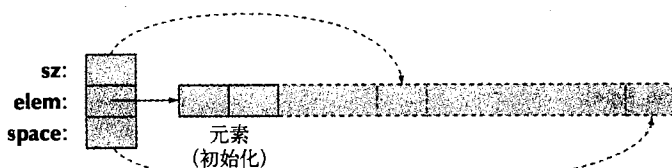
我们还是使用指向元素类型的指针 T* 作为迭代器的类型, 这是最简单的方法。我们将具有边界检查功能的迭代器的实现作为练习 (参见习题 18)。

通常情况下,人们不会对数据采用连续存储的数据类型(比如 `vector`)提供列表操作,比如 `insert()` 或 `erase()`。但是像 `insert()` 和 `erase()` 这样的列表操作对小的 `vector` 是非常有用的。我们前面已经了解了 `push_back()` 的作用,它其实也是一个列表相关的操作。

我们可以通过复制所有位于所删除元素之后的元素来实现我们的 `vector<T>::erase()`。利用 19.3.6 节中的 `vector` 定义,我们得到:

```
template<class T, class A>
vector<T,A>::iterator vector<T,A>::erase(iterator p)
{
    if (p==end()) return p;
    for (iterator pos = p+1; pos!=end(); ++pos)
        *(pos-1) = *pos;    // copy element "one position to the left"
    alloc.destroy(&*(end()-1); // destroy surplus copy of last element
    --sz;
    return p;
}
```

下图可以很好地帮助我们理解上面的代码:



`erase()` 操作的代码实现很简单,不妨在纸上试几个例子。有没有对空 `vector` 作相应的处理?为什么要判断 `p == end()`? 如果把 `vector` 的最后一个元素删除会怎么样? 如果使用索引来表示代码的可读性会不会更好?

相对而言, `vector<T, A>::insert()` 就有一点复杂了:

```
template<class T, class A>
vector<T,A>::iterator vector<T,A>::insert(iterator p, const T& val)
{
    int index = p-begin();
    if (size()==capacity()) reserve(size()==0?8:2*size()); // make sure we
                                                            // have space
    // first copy last element into uninitialized space:
    alloc.construct(elem+sz, *back());
    ++sz;
    iterator pp = begin()+index;    // the place to put val
    for (iterator pos = end()-1; pos!=pp; --pos)
        *pos = *(pos-1);    // copy elements one position to the right
    *(begin()+index) = val;    // "insert" val
    return pp;
}
```

注意:

- 由于迭代器不能指向序列之外,所以我们使用指针来完成,比如 `elem + sz`。这就是为什么配置器会使用指针进行定义而不用迭代器。
- 当使用 `reserve()` 时,元素会被移动到一块新的内存中。因此,我们必须记录所删除元素的索引值,而不是指向它的迭代器。当 `vector` 重新分配它的元素时, `vector` 的迭代器会失效——可以理解为它们指向的是旧的内存。
- 使用 `A` 作为分配器的参数很直观,但不准确。当你需要实现一个容器时,最好还是要仔细阅读一下相关的标准。

- 我们会尽量不去和底层的内存打交道。标准库中的 `vector` (包括标准库中的其他容器) 都是根据这一标准而实现的。这也是为什么我们应该尽量使用标准库的原因。

如果考虑性能的因素, 我们不会在一个含有 100 000 个元素的 `vector` 中使用 `insert()` 或 `erase()`。在这种情况下, 使用 `list` (或 `map`, 参见 21.6 节) 更为合适。但是, `vector` 中确实提供了 `insert()` 和 `erase()` 操作, 而且如果我们只需要移动几个或几十个字的元素的话, 使用它们还是没有问题的——毕竟现在的计算机已经比较强大了 (参见习题 20)。注意不要在很少的小元素上使用 `list`。

20.9 调整内置数组达到 STL 版本的功能

我们之前总是在不停地重复嵌入数组的不足之处: 它们一不小心就会转换成指针, 它们不知道自己的大小 (参见 18.5.2 节), 等等。我们也指出了它们最大的优点: 它们近乎完美地利用了物理内存的特性。

为了综合二者之长, 我们可以创建一个具有数组优点而没有其不足的 `array` 容器, `array` 容器在一份技术报告中被引入到标准中。由于并不要求在所有实现中都包含技术报告中所定义的属性, 在你所使用的实现中可能并不包含 `array`。但是, 它的思路却是简单而有用的:

```
template <class T, int N>    // not quite the standard array
struct array {
    typedef T value_type;
    typedef T* iterator;
    typedef T* const_iterator;
    typedef unsigned int size_type;    // the type of a subscript

    T elems[N];
    // no explicit construct/copy/destroy needed

    iterator begin() { return elems; }
    const_iterator begin() const { return elems; }
    iterator end() { return elems+N; }
    const_iterator end() const { return elems+N; }

    size_type size() const;

    T& operator[](int n) { return elems[n]; }
    const T& operator[](int n) const { return elems[n]; }

    const T& at(int n) const;    // range-checked access
    T& at(int n);                // range-checked access

    T* data() { return elems; }
    const T* data() const { return elems; }
};
```

上面给出的这个定义并不完整, 也不完全符合标准的要求, 但从中我们可以看出它的基本思想。如果你所使用的实现中没有包含标准 `array`, 那不妨就把它当做 `array` 的定义来使用。如果有的话, 它应该在 `<array>` 中。注意由于 `array<T, N>` “知道”它的大小为 `N`, 我们可以提供赋值、`==`、`!=` 等操作, 就像 `vector` 一样。

举例来说, 我们把 `array` 和 20.4.2 节中的 `high()` 结合起来使用:

```
void f()
{
    array<double,6> a = { 0.0, 1.1, 2.2, 3.3, 4.4, 5.5 };
    array<double,6>::iterator p = high(a.begin(), a.end());
    cout << "the highest value was " << *p << endl;
}
```


注意, 当使用 `high()` 时, 我们并没有考虑 `array`。之所以可以在 `array` 中使用 `high()`, 是因为这二者都是按照标准的习惯进行定义的。

20.10 容器概览

STL 提供了一些容器:

标准容器	
<code>vector</code>	一系列空间连续的元素; 可以用作默认容器
<code>list</code>	一个双向链表; 当希望在不移动现有元素的情况下完成对元素的插入和删除时使用
<code>deque</code>	列表和向量的结合; 除非对算法和计算机结构非常精通, 否则不要使用它
<code>map</code>	一个平衡的有序树; 当需要实现按值访问元素时使用(参见 21.6.1 ~ 21.6.3 节)
<code>multimap</code>	一个平衡的有序树, 其中可以包含同一个 <code>key</code> 的多个拷贝; 当需要实现按值访问元素时使用(参见 21.6.1 ~ 21.6.3 节)
<code>unordered_map</code>	一个散列表; 一种优化的 <code>map</code> ; 当对性能要求很高且可以设计出较好的散列函数时使用(21.6.4 节)
<code>unordered_multimap</code>	一个可以包含同一个 <code>key</code> 的多个拷贝的散列表; 一种优化的 <code>multimap</code> ; 当对性能要求很高且可以设计出较好的散列函数时使用(参见 21.6.4 节)
<code>set</code>	一个平衡的有序树; 当需要对每个值进行跟踪时使用(参见 21.6.5 节)
<code>multiset</code>	一个可以包含同一个 <code>key</code> 的多个拷贝的平衡的有序树; 当需要对每个值进行跟踪时使用(参见 21.6.5 节)
<code>unordered_set</code>	与 <code>unordered_map</code> 相似, 但只有 <code>value</code> , 不是二元组(<code>key</code> , <code>value</code>)
<code>unordered_multiset</code>	与 <code>unordered_multimap</code> 相似, 但只有 <code>value</code> , 不是二元组(<code>key</code> , <code>value</code>)
<code>array</code>	一个大小固定的数组, 不存在嵌入数组所存在的大部分问题(参见 20.6 节)

有很多关于这些容器及其使用的其他资料(书籍或网上资源)。下面给出一些比较好的参考资料:

- Austern, Matt, ed. "Technical Report on C++ Standard Library Extensions," ISO/IEC PDTR 19768. (Colloquially known as TR1.)
- Austern, Matthew H. *Generic Programming and the STL*. Addison-Wesley, 1999. ISBN 0201309564.
- Koenig, Andrew, ed. *The C++ Standard*. Wiley, 2003. ISBN 0470846747. (Not suitable for novices.)
- Lippman, Stanley B., Josée Lajoie, and Barbara E. Moo. *The C++ Primer*. Addison-Wesley, 2005. ISBN 0201721-481. (Use only the 4th edition.)
- Musser, David R., Gillmer J. Derge, and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library, Second Edition*. Addison-Wesley, 2001. ISBN 0201379236.
- Stroustrup, Bjarne. *The C++ Programming Language*. Addison-Wesley, 2000. ISBN 0201700735.
- The documentation for the SGI implementation of the STL and the `iostream` library: www.sgi.com/tech/stl. Note that they also provide complete code.
- The documentation of the Dinkumware implementation of the standard library: www.dinkumware.com/manuals/default.aspx. (Beware of several library versions.)
- The documentation of the Rogue Wave implementation of the standard library: www2.roguewave.com/support/docs/index.cfm.

你觉得被骗了吗? 你觉得我们应该对所有的容器及其应用都加以介绍? 这是不可能的。相关的标准应用、技术以及库实在是太多了, 你不可能一下就全部掌握。程序设计是一门非常广阔而多样的技术, 同时也是一门高雅的艺术。作为一名程序员, 你需要能够自己找到关于语言应用、库以及相关技术的信息。程序设计是一个时刻变化的领域, 所以如果你对自己目前所掌握的技术感到满意而就此停止, 那你很快就会变得落后。“查找相关的内容”是针对许多实际问题的一个很

好的答案，而且随着你所掌握技术的增加，它会变得越来越重要。

另一方面，你会发现当你对 `vector`、`list` 以及 `map` (见 21 章) 有了比较深入的了解后，学习其他 STL 或像 STL 一样的容器的使用会变得非常容易，你还会发现其他非 STL 容器及其应用也会变得比较容易理解了。

那么什么是容器呢？在上面给出的所有参考资料中你都可以找到关于 STL 容器的定义。下面我们只给出它的概念上的定义。一个 STL 容器

- 是一个元素序列 `[begin():end())`。
- 容器的操作可以复制元素。复制可以通过赋值或拷贝函数来实现。
- 可以确定所存元素的类型 `value - type`。
- 有迭代器 `iterator` 和 `const_iterator`。迭代器提供了 `*`、`++` (前缀和后缀) 以及 `!=` 操作符及其相应语法。`list` 的迭代器提供了可以在序列中向后移动的 `--` 操作，它也叫做双向迭代器。`vector` 的迭代器提供了 `--`、`[]`、`+` 以及 `-` 操作，它也叫做随机访问迭代器 (参见 20.10.1 节)。
- 提供了 `insert()`、`erase()`、`front()`、`back()`、`push_back()`、`pop_back()` 以及 `size()` 等操作；`vector` 和 `map` 还提供了下标功能 (例如操作符 `[]`)。
- 提供了对元素进行比较的比较操作符 (`==`、`!=`、`<`、`<=`、`>` 以及 `>=`)。容器采用字典顺序对 `<`、`<=`、`>`、`>=` 进行处理，也就是说，它们从第一个元素开始进行比较。

上述关于 STL 容器的定义可以让你了解容器大致是什么。详细的内容请参见附录 B。更详细具体的内容可以参考《The C++ Programming Language》或标准。

一些数据类型提供了标准容器所要求的一些特性，但不是全部。我们称之为“近似容器”。最常见的如下表所示：

“近似容器”	
<code>T[n]</code> 内置数组	没有 <code>size()</code> 或其他成员函数；当可以使用 <code>vector</code> 、 <code>string</code> 或 <code>array</code> 时尽量不要用内置数组
<code>string</code>	只存储字符，但对文本处理提供了许多有用的操作，比如字符串连接 (<code>+</code> 和 <code>+=</code>)；与其他字符串相比，尽量使用标准字符串
<code>valarray</code>	一个具有向量操作的数学向量，但当对性能要求较高时有许多制约，只有当需要进行大量向量计算时使用

另外，许多个人与组织都致力于符合标准容器要求的容器的开发。

如果不太确定应该采用什么样的容器，就用 `vector`。除非有充分的理由不用，否则就用 `vector`。

20.10.1 迭代器类别

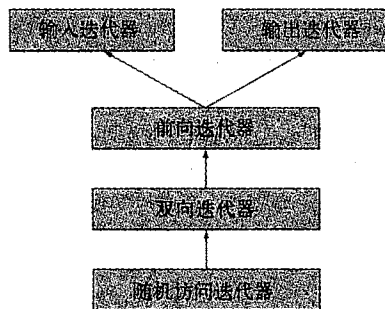
在我们之前的讨论中，似乎所有的迭代器都是可以通用的。其实，只有在进行简单的操作时它们才是通用的，比如对某个序列进行一次遍历并读取其中的数据一次。如果你希望完成更为复杂的操作，例如向后迭代和索引，你就会需要一些更为高级的迭代器了。

迭代器类别	
输入迭代器	可以利用 <code>++</code> 和 <code>*</code> 来分别完成迭代器的向前移动和元素值的读取。 <code>istream</code> 提供的就是这类迭代器，参见 21.7.2 节。如果 <code>(*p).m</code> 有效，则可以简单地使用 <code>p->m</code>
输出迭代器	可以利用 <code>++</code> 和 <code>*</code> 来分别完成迭代器的向前移动和元素值的写入。 <code>ostream</code> 提供的就是这类迭代器，参见 21.7.2 节
向前迭代器	可以利用 <code>++</code> 来完成迭代器的多次向前移动，利用 <code>*</code> 来完成对元素值的读取和写入 (除非元素是 <code>const</code> 的)。如果 <code>(*p).m</code> 有效，则可以简单地使用 <code>p->m</code>

(续)

迭代器类别	
双向迭代器	可以利用 ++ 来完成迭代器的向前移动, 利用 -- 来完成迭代器的向后移动, 利用 * 来完成对元素值的读取和写入 (除非元素是 const 的)。list、map 和 set 所提供的就是这类迭代器。如果 (*p).m 有效, 则可以简单地使用 p->m
随机访问迭代器	可以利用 ++ 来完成迭代器的向前移动, 利用 -- 来完成迭代器的向后移动, 利用 * 或 [] 来完成对元素值的读取和写入 (除非元素是 const 的)。我们可以对迭代器索引, 并利用 + 来使迭代器加上一个整数, 也可以利用 - 来使迭代器减去一个整数。我们还可以通过对两个指向同一个序列的两个迭代器进行减法操作来判断这两个迭代器之间的距离。vector 提供的就是这类迭代器。如果 (*p).m 有效, 则可以简单地使用 p->m

从这些提供的操作中可以看出, 每当使用输出或输入迭代器时, 我们都可以利用向前迭代器来完成相同的功能。双向迭代器也是一种向前迭代器, 而随机访问迭代器也是一种双向迭代器。迭代器的类别可以表示为如右图所示。注意, 由于迭代器类别并不是类, 上述层次结构并不是采用派生实现的类的层次结构。



简单练习

1. 定义一个含有 10 个元素 {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} 的 int 型数组。
2. 定义一个含有这 10 个元素的 vector <int>。
3. 定义一个含有这 10 个元素的 list <int>。
4. 另外再定义一个数组、一个向量、一个列表, 并分别把它们初始化为上面所定义的数组、向量和列表的拷贝。
5. 把数组中的每个元素值加 2; 把向量中的每个元素值加 3; 把列表中的每个元素值加 5。
6. 编写一个 copy() 函数,

```
template<class Iter1, class Iter2> Iter2 copy(Iter1 f1, Iter1 e1, Iter2 f2);
```

该操作像标准库中的 copy 函数一样把 [f1, e1) 复制到 [f2, f2 + (e1 - f1))。注意, 如果 f1 == e1, 那么该序列为空, 此时不需要复制任何内容。

7. 使用上面所定义的 copy 函数把数组中的内容复制到向量中, 再把列表中的内容复制到数组中。
8. 利用标准库中的 find() 函数来判断向量中是否含有元素 3, 如果有则输出它的位置; 利用 find() 来判断列表中是否含有元素 27, 如果有则输出它的位置。注意第一个元素的位置为 0, 第二个元素的位置为 1, 以此类推。如果 find() 函数返回的是序列的末端, 则说明没有找到所查的元素。

思考题

1. 为什么不同人编写的代码看起来会不一样? 请举例说明。
2. 我们会有哪些关于数据的简单问题?
3. 存储数据有哪些不同的方式?
4. 我们可以对一组数据做哪些基本操作?
5. 关于我们存储数据的方法有什么想法?
6. 什么是 STL 序列?
7. 什么是 STL 迭代器? 它支持哪些操作?
8. 如何把迭代器移到下一个元素?
9. 如何把迭代器移到上一个元素?
10. 当你试图把迭代器移动到序列末端之后会出现什么情况?
11. 哪些迭代器可以移动到上一个元素?

12. 为什么要把数据与算法分离开?
13. 什么是 STL?
14. 什么是链表? 它和向量本质上的区别是什么?
15. 链表中的“链”是什么?
16. `insert()` 的功能是什么? `erase()` 呢?
17. 如何判断一个序列是否为空?
18. `list` 中的迭代器提供了哪些操作?
19. 如何在 STL 容器中使用迭代器?
20. 什么时候应该使用 `string` 而不是 `vector`?
21. 什么时候应该使用 `list` 而不是 `vector`?
22. 什么是容器?
23. 容器中的 `begin()` 和 `end()` 有什么用?
24. STL 提供了哪些容器?
25. 什么是迭代器的种类? STL 提供了哪些迭代器?
26. 哪些操作随机访问迭代器提供了而双向迭代器没有提供?

术语

算法	空序列	序列	array 容器
<code>end()</code>	单链表	<code>begin()</code>	<code>erase()</code>
<code>size_type</code>	容器	<code>insert()</code>	STL
连续的	迭代	<code>typedef</code>	双向链表
迭代器	<code>value_type</code>	元素	链表

习题

1. 如果觉得还没有完全掌握的话, 请完成本章中所有的“试一试”练习。
2. 运行 20.1.2 节中的例子 Jack-and-Jill。利用几个小文件完成输入。
3. 仿照 18.6 节中回文的例子, 重新编写 20.1.2 节中的 Jack-and-Jill 程序。
4. 利用 STL 找到并修改 20.3.1 节中例子 Jack-and-Jill 中的错误。
5. 为 `vector` 定义输入和输出操作符 (`>>` 和 `<<`)。
6. 根据 20.6.2 节中的内容, 为 `Document` 编写一个查找并替换操作。
7. 查找给定 `vector<string>` 中按字典顺序最后的一个字符串。
8. 编写一个可以统计 `Document` 中字符总数的函数。
9. 编写一个可以统计 `Document` 中字数的程序。该程序有两个版本: 一种把“字”定义为“以空格分隔的字符序列”, 另一种把“字”定义为“一个连续的数字或字母序列”。例如, 在第一种定义下, `alpha number` 和 `as12b` 都是一个字, 而在第二种定义下它们则都为两个字。
10. 编写另一个统计字的程序。在该程序中, 用户可以自己定义所采用的空白字符集合。
11. 以 `list<int>` 为参数(引用调用), 创建一个 `vector<double>`, 并把列表中的元素复制到向量中。验证复制操作的完整性与正确性。然后把元素按照升序排序并打印。
12. 完成 20.4.1 节中关于 `list` 的定义并运行 `high()`。设置一个 `Link` 来表示末端后一个位置。
13. 实际上, 在 `list` 中我们并不需要一个指向末端后一个位置的 `Link`。改写上面的程序, 用 0 来代表指向末端后一个位置的 `Link(list<Elem>::end())`, 这样可以使空列表的大小和一个指针的大小相同。
14. 采用 `std::list` 的方式定义一个单向链表 `slist`。由于 `slist` 中没有后向指针, `list` 中的哪些操作可以在 `slist` 中删去?
15. 定义一个与指针 `vector` 很相似的 `pvector`。不同之处在于 `pvector` 中的指针指向对象, 而且它的析构函数 `delete` 会删除这些对象。
16. 定义一个与 `pvector` 很相似的 `ovector`。不同之处在于 `ovector` 中的 `[]` 和 `*` 操作符返回的是一个由元素(而

不是指针)所指向的对象的引用。

17. 定义一个 `ownership_vector`, 其中存储的是指向像 `pvector` 这样的对象的指针。用户可以自己定义该 `vector` 所指向的对象类型(例如,析构函数 `delete` 会删除哪些对象)。提示:如果你对第 13 章的内容比较熟悉的话就会觉得这道题比较容易了。
18. 定义一个 `vector` 中的边界检查迭代器(随机访问迭代器)。
19. 定义一个 `list` 中的边界检查迭代器(双向迭代器)。
20. 对使用 `vector` 和 `list` 的时间消耗做一个小实验。有关如何对程序进行计时的内容可以在 26.6.1 节中找到。生成 N 个属于区间 $[0:N)$ 的 `int` 型随机数。每生成一个随机数就把它加入到 `vector<int>` 中(该 `vector` 大小每次加 1)。保持该 `vector` 为有序的,也就是说, `vector` 中新元素之前的所有元素都小于等于新元素的值,而新元素之后的所有元素都大于新元素的值。针对 `list<int>` 完成相同的操作。在 N 取什么样的值时 `list` 会比 `vector` 快? 请给出解释说明。该实验由 John Bentley 最先提出。

附言

假设我们有 N 种容器和相关的 M 种操作,那么我们会需要 $N * M$ 段代码。如果所处理的数据有 K 种不同的类型,那么代码段的总数就会达到 $N * M * K$ 。STL 通过把数据类型作为参数(解决了因子 K 的问题)和把算法与对数据的访问分离开解决了这一问题。通过利用迭代器实现对不同容器不同算法中数据的访问,我们只需要 $N + M$ 种算法。这大大简化了我们的工作。举例来说,如果我们有 12 种容器和 60 种算法,那么我们会需要 720 个函数;而 STL 只需要 60 个函数和 12 种迭代器,这可以省去我们 90% 的工作量。实际上,这还是保守估计,因为很多算法处理两对迭代器,而它们不必是相同类型(参见习题 6)。而且,STL 还提供了可以很简单地对算法代码进行编写和组合定义的方法,这会使我们的工作得到进一步简化。

第 21 章 算法和映射

“理论上，实践是简单的。”

——Trygve Reenskaug

本章将完善我们对 STL 基本思想以及 STL 所提供工具的介绍。本章的主要目的是帮助你学习一些最有用的 STL 工具，它们能够帮助你节约大量的代码开发时间。我们将通过实例的方式介绍每一种工具的用途以及它所采用的编程技术。本章的另一个目的是提供足够的工具以使得你能够根据需要编写自己的不同于标准库或其他库的（优雅和有效的）算法。另外，本章还将介绍三种容器：map、set 和 unordered_map。

21.1 标准库中的算法

标准库大约提供了 60 种有用的算法。我们将在本章着重介绍其中一些最常用的算法以及在某些场合中十分有用的一些算法：

标准算法	
<code>r = find(b, e, v)</code>	<code>r</code> 指向在 <code>[b:e)</code> 中 <code>v</code> 首次出现的位置
<code>r = find_if(b, e, p)</code>	<code>r</code> 指向在 <code>[b:e)</code> 中使得 <code>p(x) == true</code> 的第一个元素 <code>x</code>
<code>x = count(b, e, v)</code>	<code>x</code> 为 <code>v</code> 在 <code>[b:e)</code> 中出现的次数
<code>x = count_if(b, e, p)</code>	<code>x</code> 为在 <code>[b:e)</code> 中能够满足条件 <code>p(x) == true</code> 的元素个数
<code>sort(b, e)</code>	通过 <code><</code> 操作符对 <code>[b:e)</code> 排序
<code>sort(b, e, p)</code>	通过 <code>p</code> 对 <code>[b:e)</code> 排序
<code>copy(b, e, b2)</code>	将 <code>[b:e)</code> 拷贝至 <code>[b2:b2 + (e - b))</code> ； <code>b2</code> 之后应有足够的用于存储元素的空间
<code>unique_copy(b, e, b2)</code>	将 <code>[b:e)</code> 拷贝至 <code>[b2:b2 + (e - b))</code> ；算法不拷贝相邻的重复元素
<code>merge(b, e, b2, e2, r)</code>	将有序序列 <code>[b2:e2)</code> 和 <code>[b:e)</code> 合并，并放入 <code>[r:r + (e - b) + (e2 - b2))</code> 之中
<code>r = equal_range(b, e, v)</code>	<code>r</code> 是有序范围 <code>[b:e)</code> 中值为 <code>v</code> 的子范围，本质上，就是对 <code>v</code> 的二分搜索
<code>equal(b, e, b2)</code>	判断 <code>[b:e)</code> 元素和 <code>[b2:b2 + (e - b))</code> 的元素是否相等
<code>x = accumulate(b, e, i)</code>	<code>x</code> 是将 <code>i</code> 与 <code>[b:e)</code> 中所有元素进行累加的结果
<code>x = accumulate(b, e, i, op)</code>	与 <code>accumulate</code> 类似，但通过 <code>op</code> 进行“求和”运算
<code>x = inner_product(b, e, b2, i)</code>	<code>x</code> 是 <code>[b:e)</code> 与 <code>[b2:b2 + (e - b))</code> 的内积
<code>x = inner_product(b, e, b2, i, op, op2)</code>	与 <code>inner_product</code> 类似，但通过 <code>op</code> 和 <code>op2</code> 取代内积的 <code>+</code> 和 <code>*</code> 操作

默认情况下，相等关系通过 `==` 操作判断，而排序则通过 `<`（小于）操作进行。标准库算法在 `<algorithm>` 头文件中声明。如果读者想获得更多信息，请参考附录 B.5 和 20.7 节中的资源列表。这些算法能够处理一个或几个序列。一个输入序列由一对迭代器定义；一个输出序列由一个指向首元素的迭代器定义。通常，一种算法可以由一个或多个操作进行参数化，其中这些操作可以是函数对象或函数。这些算法通常会通过返回输入序列的结尾作为任务“失败”的报告。例如，`find(b, e, v)` 返回 `e`，如果它未找到 `v`。

21.2 最简单的算法：find()

`find()` 是最简单但十分有用的一种算法，它的用途是在一个序列中查找一个给定值：


```

template<class In, class T>
In find(In first, In last, const T& val)
// find the first element in [first,last) that equals val
{
    while (first!=last && *first != val) ++first;
    return first;
}

```

让我们看看上述 `find()` 的定义。你可以在编程中使用 `find()` 而并不需要了解它的具体实现——实际上，我们已经在前面的章节中使用过 `find()` 了（例如 20.6.2 节）。尽管如此，`find()` 函数的定义包含了很多有用的设计思想，因此了解它的实现是有价值的。

首先，`find()` 对由一对迭代器定义的序列进行操作。`find()` 函数在半开区间 `[first:last)` 中查找给定值 `val`，而函数的返回结果为一个迭代器。返回结果要么指向在序列中 `val` 首次出现的位置，要么返回 `last`。在 STL 中，返回一个指向序列中紧接末尾元素之后位置的迭代器通常用于表示“未找到”。因此，我们可以采用如下方式使用 `find()`：

```

void f(vector<int>& v, int x)
{
    vector<int>::iterator p = find(v.begin(), v.end(), x);
    if (p!=v.end()) {
        // we found x in v
    }
    else {
        // no x in v
    }
    // ...
}

```

在上面的例子中，序列由一个容器（STL `vector`）所包含的所有元素组成。我们检查返回的迭代器是否指向序列的末端，以判断 `find()` 是否找到了我们想要的值。

到目前为止，我们已经学会了如何使用 `find()`，从而也了解了如何使用那些采用相似用法的算法。在学习更多用法和算法之前，让我们再进一步观察 `find()` 的定义：

```

template<class In, class T>
In find(In first, In last, const T& val)
// find the first element in [first,last) that equals val
{
    while (first!=last && *first != val) ++first;
    return first;
}

```

当你第一次看这段代码时，你会注意代码中的循环吗？这一循环的实现实际上是简洁高效的，且它是表示基本算法的一种直接方式。然而，你很可能在代码中不会注意到这一循环。现在，让我们用比较常见的方式重新实现 `find()` 函数，并对这两种实现版本进行比较：

```

template<class In, class T>
In find(In first, In last, const T& val)
// find the first element in [first,last) that equals val
{
    for (In p = first; p!=last; ++p)
        if (*p == val) return p;
    return last;
}

```

这两种定义在逻辑上是等价的，且一个优秀的编译器能够为这两种定义生成相同的底层代码。然而，在现实中，很多编译器都不具有这种能力，它们无法消除额外的变量（`p`），也不能对代码进行重排以使所有的条件测试都能够在同一个位置被执行。那么我们为什么要担忧和进行解释呢？

一部分原因在于 `find()` 的第一种(完美的)定义版本的风格已变得十分流行,我们必须学会它以阅读其他用户的代码;另一部分原因是,对于用于处理大量数据的、短小且被频繁使用的函数而言,其性能是十分重要的。

试一试 你能保证上述两种定义在逻辑上是等价的吗?你是如何保证的?试着提供证据证明二者是等价的。然后,通过数据测试这两种定义。著名的计算机科学家(Don Knuth)曾经说,“我只是证明了算法的正确性,但并没有对它进行测试”。即使是数学证明都可能包含错误。为了证明你的观点,进行推理和测试缺一不可。

21.2.1 一些一般的应用

`find()` 算法是泛型的、通用的。这意味着,这一算法能够被用于不同的数据类型。实际上, `find()` 算法的通用性包括两个方面;它能用于:

- 任何 STL 风格的序列
- 任何元素类型

下面是一些例子(如果你感到困惑,请参考 20.4 节中的图表):

```
void f(vector<int>& v, int x)    // works for vector of int
{
    vector<int>::iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* we found x */}
    // ...
}
```

在上面的例子中, `find()` 使用了 `vector<int>::iterator` 的迭代操作;也就是说, `++(++first)` 只是将指针指向内存中的下一个位置(存储了 `vector` 的下一个元素),且 `*` (在 `*first` 中)对该指针进行解引用。迭代器之间的比较(在 `first!=last` 中)是指针的比较,且对迭代器的取值进行比较(在 `*first!=val` 中)则只是对两个整数进行比较。

```
void f(list<string>& v, string x)    // works for list of string
{
    list<string>::iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* we found x */}
    // ...
}
```

在上面的代码中, `find()` 使用了 `vector<string>::iterator` 的迭代操作。这个例子具有与上面的 `vector<int>` 例子相同的逻辑。尽管如此,代码具体实现是不同的;也就是说, `++` 操作(`++first`)将使指针顺着元素的 `Link` 指针部分指向 `list` 下一元素的存储位置,而 `*` (在 `*first` 中)操作将获得 `Link` 的数据部分。迭代器之间的比较(在 `first!=last` 中)是 `Link*` 类型指针的比较,而对迭代器的取值进行比较(在 `*first!=val` 中)则是通过 `string` 类型的 `!=` 操作符比较两个字符串。

因此, `find()` 函数是相当灵活的:只要遵循了迭代器的规则,我们就可以通过 `find()` 对我们指定的序列和定义的容器进行查找。例如,可以使用 `find()` 在 `Document` (参见 20.6 节中的定义)中查找一个字符:

```
void f(Document& v, char x)    // works for Document of char
{
    Text_iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* we found x */}
    // ...
}
```

这种类型的灵活性是 STL 算法的特点,它使得 STL 算法具有十分强大的功能。

21.3 通用搜索算法: find_if()

在实际中,我们并不需要经常查找一个特定值。更常见的情况是,我们对在序列中查找符合某些标准的值更感兴趣。如果我们能够在查找中定义自己的查找标准,那么我们就能够得到一个更为有用的 find 操作。例如,我们也许希望在序列中查找大于 42 的值。我们也许希望在不考虑大小写的情况下比较字符串。我们也许希望找到序列中的第一个奇数值。我们也许希望查找一个地址域值为“17 Cherry Tree Lane”的记录。

根据用户提供的标准进行查找的标准算法是 find_if():

```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
{
    while (first!=last && !pred(*first)) ++first;
    return first;
}
```

显然(当你对源代码进行比较时), find_if() 的实现与 find() 的实现很相似,除了前者使用 !pred(*first) 而非 *first != val; 也就是说,一旦谓词 pred() 成立,则 find_if() 立即停止查找。

谓词是一种返回 true 或 false 的函数。find_if() 要求谓词具有一个参数,以使得它能够做出判断 pred(*first)。我们可以比较容易地编写一个谓词以检查一个给定值的属性,例如“给定字符串是否包含字母 x?”“给定值是否大于 42?”“给定数是否是奇数?”例如,我们可以通过如下方式在 int 型的向量中查找第一个奇数:

```
bool odd(int x) { return x%2; } // % is the modulo operator

void f(vector<int>& v)
{
    vector<int>::iterator p = find_if(v.begin(), v.end(), odd);
    if (p!=v.end()) { /* we found an odd number */ }
    // ...
}
```

find_if() 对每一元素调用 odd() 直至它找到了第一个奇数。注意,当你将一个函数作为参数传递时,不要在其名字后面加上(),否则传递的将不是函数,而是其调用结果。

类似地,我们可以找到一个列表中第一个大于 42 的元素:

```
bool larger_than_42(double x) { return x>42; }

void f(list<double>& v)
{
    list<double>::iterator p = find_if(v.begin(), v.end(), larger_than_42);
    if (p!=v.end()) { /* we found a value > 42 */ }
    // ...
}
```

上面的例子并不是十分令人满意。如果下次我们想找出大于 41 的元素该怎么办呢?我们不得不编写一个新的函数。那查找大于 19 的元素又该如何?还要编写另一个函数。应该有更好的方法!

如果我们想要与任意的值 v 进行比较,我们需要使 v 能够成为 find_if() 谓词的一个隐含参数。我们可以编写如下形式的代码(选择 v_val 作为变量名以避免与其他名称发生冲突):

```
double v_val; // the value to which larger_than_v() compares its argument
bool larger_than_v(double x) { return x>v_val; }

void f(list<double>& v, int x)
```

```

{
    v_val = 31; // set v_val to 31 for the next call of larger_than_v
    list<double>::iterator p = find_if(v.begin(), v.end(), larger_than_v);
    if (p!=v.end()) { /* we found a value > 31 */}

    v_val = x; // set v_val to x for the next call of larger_than_v
    list<double>::iterator q = find_if(v.begin(), v.end(), larger_than_v);
    if (q!=v.end()) { /* we found a value > x */}

    // ...
}

```

我们能够保证上述代码能够实现我们想要的功能，但是上述代码维护困难。再次地，我们认为应该还有更好的方法！

试一试 为什么我们不愿意按照上述方式使用 `v` 呢？给出由上述方式所可能产生的三个错误。列出三个你特别讨厌出现上述错误的应用程序。

21.4 函数对象

我们希望能够向 `find_if()` 传递谓词，同时我们也希望谓词能够将元素与以参数形式传递的值进行比较。特别地，我们希望能编写如下形式的代码：

```

void f(list<double>& v, int x)
{
    list<double>::iterator p = find_if(v.begin(), v.end(), Larger_than(31));
    if (p!=v.end()) { /* we found a value > 31 */}

    list<double>::iterator q = find_if(v.begin(), v.end(), Larger_than(x));
    if (q!=v.end()) { /* we found a value > x */}

    // ...
}

```

显然，`Larger_than` 必须满足以下条件：

- `Larger_than` 能够以断言的形式被调用，例如 `pred(*first)`。
- 能够存储一个数值，例如 31 或 `x`，供被调用时使用。

为了满足这些条件，我们需要“函数对象”，即一种能够实现函数功能的对象。我们需要对象是因为对象能够存储数据，例如待比较的值。例如：

```

class Larger_than {
    int v;
public:
    Larger_than(int vv) : v(vv) { } // store the argument
    bool operator()(int x) const { return x>v; } // compare
};

```

有趣的是，上述定义能够使上面的例子正常工作。现在，我们需要指出为什么这一定义具有这样的作用。对于 `Larger_than(31)`，它代表一个 `Larger_than` 类的对象，且其数据成员 `v` 的取值为 31，例如：

```
find_if(v.begin(),v.end(),Larger_than(31))
```

在上面的代码中，我们将对象 `Larger_than(31)` 作为参数 `pred` 的实参传递给 `find_if()`。对于 `v` 的每一个元素，`find_if()` 调用：

```
pred(*first)
```

`pred(*first)` 将会调用我们的函数对象 `Larger_than(31)` 的调用操作符，即 `operator()`。因此，

调用的结果是将是元素值 *first 和 31 的比较结果。

在这里,我们发现上述函数调用可以视为一个操作符“()操作符”。“()操作符”也称为函数调用操作符和应用操作符。因此, `pred(*first)` 中的 `()` 代表了操作符 `Larger_than::operator()`, 就像 `v[i]` 中的下标代表了操作符 `vector::operator[]` 一样。

21.4.1 函数对象的抽象视图

我们已经学习了一种机制,这一机制允许一个“函数”能够“承载”它所要的数据。函数对象为我们提供了一种通用的、强大的、便利的机制。下面的例子展示了函数对象的更一般性的概念:

```
class F {      // abstract example of a function object
    S s; // state
public:
    F(const S& ss) : s(ss) { /* establish initial state */ }
    T operator() (const S& ss) const
    {
        // do something with ss to s
        // return a value of type T (T is often void, bool, or S)
    }

    const S& state() const { return s; } // reveal state
    void reset(const S& ss) { s = ss; } // reset state
};
```

一个 F 类的对象通过其成员 s 存储数据。根据需要,一个函数对象能够拥有很多数据成员。当一个对象存储了数据之后,则我们可以称该对象具有了“状态”。当构造 F 类的对象时,我们可以初始化其状态。我们也能够根据需要读取该对象的状态。对于 F 类,我们实现了操作 `state()` 以读取 F 类对象的状态,操作 `reset()` 以设置 F 类对象的状态。然而,当设计一个函数对象时,我们能够根据自己的需要实现访问对象状态的方法。同时,我们也可以直接或间接地通过普通函数调用的概念调用函数对象。在上述代码中,我们将 F 在被调用时只接收一个参数,但我们可以具有多个参数的函数对象。

函数对象的用法代表了 STL 中参数化的主要方法。我们可以通过函数对象指定需要查找的对象(参见 21.3 节),定义排序标准(参见 21.4.2 节),在数值型算法中指定算术操作(参见 21.5 节),定义值相等的含义(参见 21.8 节)以及其他很多事情。函数对象的使用是灵活性和一般性的主要来源。

函数对象通常是十分高效的。特别地,向一个模板函数以传值的方式传递一个小的函数对象能够带来性能的优化。原因很简单,但对于只熟悉以函数为参数方式传递的人来说可能是奇怪的:由传递函数对象方式所产生的代码要比由传递函数方式所产生的代码更少、更快。这一结论是正确的,仅当函数对象较小(如只占 0、1 或 2 个字)或者函数对象通过引用方式传递,并且函数调用操作符的操作比较简单(如简单的比较操作 <),同时函数对象以内联的形式定义(例如,函数对象在其类中实现所有定义)。在本章中(本书中)的大多数例子的函数对象都满足这些条件。小且简单的函数对象能够带来高性能的基本原因在于它们保存了足够的类型信息以供编译器产生优化代码。甚至老式编译器中的优化器都能够为 `Larger_than` 中的比较操作产生简单的表示“大于”的机器指令,但它们不能为函数调用实现这一优化。函数调用所需花费的时间通常是执行简单比较操作所花费时间的 10~50 倍。另外,函数调用所形成的代码通常是简单比较操作所形成的代码的几倍。

21.4.2 类成员上的谓词

如我们刚才所见，标准算法能够正确处理由基本类型的元素组成的序列，如 `int` 和 `double`。然而，在一些应用领域，类对象的容器更为常见。下面是一个根据一些标准对记录进行排序的例子：

```
struct Record {
    string name;      // standard string for ease of use
    char addr[24];    // old style to match database layout
    // ...
};

vector<Record> vr;
```

我们有时希望能够根据名字进行排序，有时又希望能够根据地址进行排序。如果不能够同时实现这两种排序标准，那么我们的代码很可能会没有什么实用价值。幸运的是，同时实现两种排序标准并不难。我们可以编写如下代码：

```
// ...
sort(vr.begin(), vr.end(), Cmp_by_name()); // sort by name
// ...
sort(vr.begin(), vr.end(), Cmp_by_addr()); // sort by addr
// ...
```

`Cmp_by_name` 函数对象根据 `Record` 的 `name` 成员对两个 `Record` 对象进行排序。`Cmp_by_addr` 函数对象根据 `Record` 的 `addr` 成员对两个 `Record` 对象进行排序。为了使用户能够指定比较标准，标准库算法 `sort` 采用了可选的第三参数以供用户指定比较标准。`Cmp_by_name()` 构造了一个 `Cmp_by_name` 对象。现在，我们需要定义 `Cmp_by_name` 和 `Cmp_by_addr`：

// different comparisons for Record objects:

```
struct Cmp_by_name {
    bool operator()(const Record& a, const Record& b) const
    { return a.name < b.name; }
};

struct Cmp_by_addr {
    bool operator()(const Record& a, const Record& b) const
    { return strcmp(a.addr, b.addr, 24) < 0; } // !!!
};
```

`Cmp_by_name` 类的实现十分简单。函数调用操作符，`operator()()` 仅仅通过标准 `string` 的 `<` 操作符对 `name` 字符串进行比较。然而，`Cmp_by_addr` 中的比较操作实现得并不好。这是因为采用了一种较差的方式表示地址：由 24 个字符组成的数组（非 0 结尾）。之所以采用这一方式，一部分原因在于展示函数对象是如何用于掩盖丑陋且容易产生错误的代码的。另一部分原因是这一特别的表示方式曾被我认为是一个挑战：“一个不能通过 STL 处理的丑陋而又重要的现实问题。”实际上，STL 能够处理。比较函数使用了标准 C（和 C++）库函数 `strcmp()`，该函数能够对固定长度的字符数组进行比较，并且当第二个“字符串”在字典顺序上排在第一个“字符串”之后时，它将返回一个负数（参考附录 B.10.3）。

21.5 数值算法

大多数的标准库算法都涉及处理数据管理问题：它们需要对数据进行拷贝、排序、查找等。然而，只有少数算法涉及数值计算。当我们需要进行计算时，这些数值算法就变得十分重要了，并且这些算法为我们在 STL 框架中编写数值算法提供了启示。

在 STL 标准库中只有 4 种数值算法：

数值算法

<code>x = accumulate(b, e, i)</code>	累加序列中的值；例如，对序列 <code> a, b, c, d </code> 进行 <code>a + b + c + d</code> 计算。结果 <code>x</code> 的类型与初始值 <code>i</code> 的类型一致
<code>x = inner_product(b, e, b2, i)</code>	将两个序列的对应元素相乘并将结果累加。例如，对序列 <code> a, b, c, d </code> 和 <code> e, f, g, h </code> 进行 <code>a * e + b * f + c * g + d * h</code> 计算。结果 <code>x</code> 的类型与初始值 <code>i</code> 的类型一致
<code>r = partial_sum(b, e, r)</code>	对一个序列的前 <code>n</code> 个元素进行累加，并根据每次累加的结果生成一个序列。例如，对序列 <code> a, b, c, d </code> 进行操作将生成序列 <code> a, a + b, a + b + c, a + b + c + d </code>
<code>r = adjacent_difference(b, e, b2, r)</code>	对一个序列的相邻元素进行减操作，并根据每次的结果生成一个序列。例如，对序列 <code> a, b, c, d </code> 进行操作将生成序列 <code> a, b - a, c - b, d - c </code>

这些算法可以在 `<numeric>` 中找到。我们将介绍前两个，如果有需要的话，读者可以自己查找后两个的详细情况。

21.5.1 累积

`accumulate()` 函数是最简单但最有用的数值算法。在其最简单的形式中，该算法将序列值进行累加：

```
template<class In, class T> T accumulate(In first, In last, T init)
{
    while (first != last) {
        init = init + *first;
        ++first;
    }
    return init;
}
```

给定初始值 `init`，该算法将序列 `[first: last)` 中的每一个值及 `init` 进行累加，并返回所得累加值。`init` 通常称为累加器。例如：

```
int a[] = { 1, 2, 3, 4, 5 };
cout << accumulate(a, a+sizeof(a)/sizeof(int), 0);
```

上述代码的打印结果为 15，即 $0 + 1 + 2 + 3 + 4 + 5$ 。显然，`accumulate()` 能够用于所有类型的序列：

```
void f(vector<double>& vd, int* p, int n)
{
    double sum = accumulate(vd.begin(), vd.end(), 0.0);
    int sum2 = accumulate(p, p+n, 0);
}
```

累加结果的类型与 `accumulate()` 中累加器的类型一致。这一灵活性是十分重要的，例如：

```
void f(int* p, int n)
{
    int s1 = accumulate(p, p+n, 0);           // sum into an int
    long sl = accumulate(p, p+n, long(0));    // sum the ints into a long
    double s2 = accumulate(p, p+n, 0.0);      // sum the ints into a double
}
```

在一些计算机系统中，`long` 的有效位数要比 `int` 更多。与 `int` 型相比，`double` 型能够表示的数的范围更大，但可能精度更差。我们将在第 24 章介绍在数值计算中范围和精度所起的作用。

将用于存储算法的最终结果的变量作为 `accumulate()` 的初始值是一种比较常见的用法：

```
void f(vector<double>& vd, int* p, int n)
{
    double s1 = 0;
    s1 = accumulate(vd.begin(), vd.end(), s1);
    int s2 = accumulate(vd.begin(), vd.end(), s2);    // oops
    float s3 = 0;
    accumulate(vd.begin(), vd.end(), s3);            // oops
}
```

我们应该记住初始化累加器以及使用变量保存 `accumulate()` 的结果。在上面的例子中, `s2` 在成为 `accumulate()` 初始值时并未被初始化; 因此算法的结果是不可预知的。我们将 `s3` 传递给 `accumulate()` (传值方式, 参见 8.5.3 节), 但算法结果并未保存; 因此, 算法的执行只会造成时间的浪费。

21.5.2 一般化 `accumulate()`

基本的 `accumulate()` 具有三个参数。然而, 在实际中我们可能需要使用其他有用的操作 (例如乘法和减法) 对序列进行处理。为此, STL 还提供了一个具有 4 个参数的 `accumulate()` 算法, 在这一算法中我们能够指定所要使用的操作:

```
template<class In, class T, class BinOp>
T accumulate(In first, In last, T init, BinOp op)
{
    while (first!=last) {
        init = op(init, *first);
        ++first;
    }
    return init;
}
```

任何能够处理累加器类型的参数的二元操作均能用于这一版本的 `accumulate()` 算法。例如:

```
array<double,4> a = { 1.1, 2.2, 3.3, 4.4 }; // see §20.9
cout << accumulate(a.begin(),a.end(), 1.0, multiplies<double>());
```

上述代码将打印 35.1384, 即 $1.0 * 1.1 * 2.2 * 3.3 * 4.4$ (1.0 为初始值)。代码中的二元操作符 `multiplies < double > ()` 是一个实现乘法操作的标准库函数对象; `multiplies < double >` 实现 `double` 数的乘法; `multiplies < int >` 实现 `int` 数的乘法, 等等。还有一些其他的二元函数对象: `plus` (加法)、`minus` (减法)、`divides`、`modulus` (取余)。这些对象均在 `<functional>` 中定义 (参见附录 B.6.2)。

注意, 为了对浮点数进行处理, 初始值设为 1.0。

如 `sort()` 例子 (参见 21.4.2 节) 中所示, 我们通常对类对象中包含的数据更感兴趣, 而不仅仅是内建的类型。例如, 给定物品的单价和数量, 我们可以计算所有物品的价值总和:

```
struct Record {
    double unit_price;
    int units; // number of units sold
    // ...
};
```

我们可以使 `accumulate` 的操作符能够从一个 `Record` 元素中抽取 `units`, 并计算价格及实现累加:

```
double price(double v, const Record& r)
{
    return v + r.unit_price * r.units; // calculate price and accumulate
}

void f(const vector<Record>& vr)
{
    double total = accumulate(vr.begin(), vr.end(), 0.0, price);
    // ...
}
```

我们在这里使用了函数, 而不是函数对象——我们这么做仅仅是为了展示。当下面两个条件之一能够得到满足时, 那么我们应使用函数对象:

- 如果函数对象需要在调用间保存数值, 或
- 如果函数对象能以内联形式实现

根据第二个条件, 我们应该在上面这个例子中选择函数对象的方式。

试一试 定义一个 `vector<Record>` 对象, 用你所选择物品的 4 个记录将其初始化, 并通过上面的函数计算物品的总价值。

21.5.3 内积

给定两个向量, 将它们对应位置的元素相乘并将结果累加, 这一操作称为向量的内积, 内积在很多领域都十分有用(例如, 物理和线性代数; 参见 24.6 节)。下面是 STL 版本的内积实现:

```
template<class In, class In2, class T>
T inner_product(In first, In last, In2 first2, T init)
// note: this is the way we multiply two vectors (yielding a scalar)
{
    while(first!=last) {
        init = init + (*first) * (*first2);    // multiply pairs of elements
        ++first;
        ++first2;
    }
    return init;
}
```

上面的代码能用于任何元素类型的任何序列。以股票市场指数为例, 在股票市场中, 每一个上市公司都会被分配一个“权重”。例如, 在道琼斯工业指数中, Alcoa 公司的权重为 2.4808。为了获得市场指数, 我们需要将每一个公司的股票价值与其权重相乘, 并将所得结果进行累加。显然, 这里需要进行价格和权重之间的内积操作。例如:

```
// calculate the Dow Jones Industrial index:
vector<double> dow_price;    // share price for each company
dow_price.push_back(81.86);
dow_price.push_back(34.69);
dow_price.push_back(54.45);
// ...

list<double> dow_weight;    // weight in index for each company
dow_weight.push_back(5.8549);
dow_weight.push_back(2.4808);
dow_weight.push_back(3.8940);
// ...

double dji_index = inner_product( // multiply (weight,value) pairs and add
    dow_price.begin(), dow_price.end(),
    dow_weight.begin(),
    0.0);

cout << "DJI value " << dji_index << "\n";
```

注意, `inner_product()` 需要处理两个序列。但是, `inner_product()` 只用三个参数对这两个序列进行描述: 参数只指出了第二个序列的开始位置。这一算法假设第二个序列包含的元素个数要等于或大于第一个序列。如果这一假设不成立, 则将产生错误。在我们的例子中, 这一假设是成立的; 那些“多余的”元素将不会被处理。

两个序列不需要具有相同的类型, 且序列元素的类型也不必相同。为了说明这一点, 我们使用 `vector` 存储价格而采用 `list` 存储权重。

21.5.4 一般化 `inner_product()`

`inner_product()` 也可以像 `accumulate()` 一样具有一般性, 但 `inner_product()` 需要两个额外的参数: 一个用于设置累加器(与 `accumulate()` 一样), 一个用于结合元素的值对:

```
template<class In, class In2, class T, class BinOp, class BinOp2 >
T inner_product(In first, In last, In2 first2, T init, BinOp op, BinOp2 op2)
{
    while(first!=last) {
        init = op(init, op2(*first, *first2));
        ++first;
        ++first2;
    }
    return init;
}
```

在 21.6.3 节中，我们将回到道琼斯的例子，并在代码中使用这种 inner_product() 形式。

21.6 关联容器

除了 vector 之外，最有用的标准库容器恐怕就是 map。每个 map 是一个 (key, value) 对的有序队列，使用它你可以基于一个 key 来查找一个 value；例如 my_phone_book[“Nicholas”] 可能是 Nicholas 的电话号码。在流行的竞争中，map 唯一的潜在竞争对手是 unordered_map (参见 21.6.4 节)，它是一种经过针对字符串关键字优化过的 map。map 和 unordered_map 有很多相似的数据结构，例如关联数组、散列表和红黑树等。流行的和有用的概念总是看起来有很多名称。在标准库中，我们将这类数据结构统称为关联容器。

标准库提供了如下 8 个关联容器：

关联容器	
map	(key, value) 对的有序容器
set	key 的有序容器
unordered_map	(key, value) 对的无序容器
unordered_set	key 的无序容器
multimap	key 可以出现多次的 map
multiset	key 可以出现多次的 set
unordered_multimap	key 可以出现多次的 unordered_map
unordered_multiset	key 可以出现多次的 unordered_set

这些容器可以在 < map >、< set >、< unordered_map > 和 < unordered_set > 中找到。

21.6.1 映射

思考一个概念上简单的例子：建立一个单词在文本中出现次数的列表。最明显的方式是维护一个我们看到单词的列表，我们看到每个单词后将次数加起来。当看到一个新的单词时，我们首先看是否曾经看到过它；如果我们曾经看到它，将该单词相应的计数器加 1；否则，将它插入列表并赋值为 1。我们可以使用列表或数组来完成它，但是我们不得不为读取的每个单词进行一次查找。这个过程可能是缓慢的。映射存储关键字的方式使得查找关键字是否存在十分容易，因此查找部分在我们的任务中是微不足道的：

```
int main()
{
    map<string,int> words;    // keep (word,frequency) pairs

    string s;
    while (cin>>s) ++words[s]; // note: words is subscripted by a string

    typedef map<string,int>::const_iterator Iter;
    for (Iter p = words.begin(); p!=words.end(); ++p)
        cout << p->first << ": " << p->second << '\n';
}
```


这个程序中有趣的部分实际是 `++ words[s]`。正如我们在 `main()` 第一行中看到的, `words` 是一个 `(string, int)` 对的映射;也就是说, `words` 将 `string` 映射到 `int`。换句话说,如果我们得到一个 `string`, `words` 可以让我们访问对应的 `int`。当我们用 `string`(通过输入得到单词)来标记 `words` 时, `words[s]` 是一个将 `int` 对应于 `s` 的引用。让我们来看一个具体的例子:

```
words["sultan"]
```

如果我们没看到过字符串“sultan”, “sultan”将会以默认值 0 存入 `words`。现在, `words` 有一个入口 (“sultan”, 0)。接下来,如果我们在此之前没有看到“sultan”, `++ words[“sultan”]` 会将值 1 与字符串“sultan”相关联。详细来说: `map` 发现没有找到“sultan”, 插入一个 (“sultan”, 0) 对, 然后 `++` 会将该值加 1, 得到 1。

我们现在回来看这个程序: `++ words[s]` 检测每个输入的单词, 并将它对应的值加 1。如果一个新的单词第一次出现, 它将会得到的值为 1。现在, 这个循环的含义是清晰的:

```
while (cin>>s) ++words[s];
```

这个程序读取输入的每个单词(用空格分开), 并计算每个单词的出现次数。现在我们要做的是生成输出。我们可以遍历一个映射, 就像其他 STL 容器一样。每个 `map<string, int>` 的元素是 `<string, int>` 值对。每个值对的第一个元素是 `first`, 第二个元素是 `second`, 因此输出循环为

```
typedef map<string,int>::const_iterator lter;
for (lter p = words.begin(); p!=words.end(); ++p)
    cout << p->first << ": " << p->second << '\n';
```

`typedef`(参见 20.5 节和附录 A.16)只是为了表示方便和具有可读性。

为了进行测试, 我们可以将第一个版本的《The C++ Programming Language》第 1 版的开篇加入程序:

```
C++ is a general purpose programming language designed to make pro-
gramming more enjoyable for the serious programmer. Except for minor
details, C++ is a superset of the C programming language. In addition to
the facilities provided by C, C++ provides flexible and efficient facilities
for defining new types.
```

我们得到输出

```
C: 1
C++: 3
C,: 1
Except: 1
In: 1
a: 2
addition: 1
and: 1
by: 1
defining: 1
designed: 1
details,: 1
efficient: 1
enjoyable: 1
facilities: 2
flexible: 1
for: 3
general: 1
is: 2
```

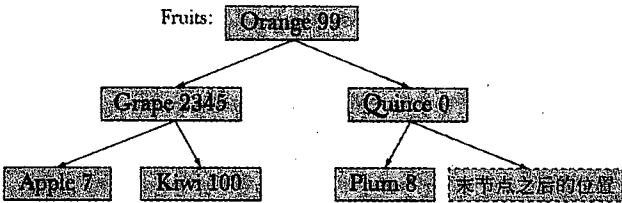
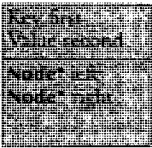
```
language: 1
language.: 1
make: 1
minor: 1
more: 1
new: 1
of: 1
programmer.: 1
programming: 3
provided: 1
provides: 1
purpose: 1
serious: 1
superset: 1
the: 3
to: 2
types.: 1
```

如果我们不想区分大小写字母或删除标点符号，我们可以这样做：参见练习 13。

21.6.2 map 概览

那么，什么是映射呢？映射的实现有很多种方式，但是 STL 映射的实现通常是平衡二叉查找树，更具体地说是红黑树。我们将不会深入讨论它，但是现在你知道了这个术语。这样，如果你想了解更多的知识，你可以通过书籍或网页进行查找。

一棵树是由多个节点（与列表由链接构成的相似；参见 20.4 节）构成。每个节点保存一个关键字，它有相关的值并指向两个子节点，如右图所示。这就是 `map < Fruit, int >` 在内存中的样子，假设我们插入了 (Kiwi, 100)、(Quince, 0)、(Plum, 8)、(Apple, 7)、(Grape, 2345) 和 (Orange, 99)：



得到持有关键字的值为 first 的节点成员的名字，二叉查找树的基本规则是：

$left \rightarrow first < first \ \&\& \ first < right \rightarrow first$

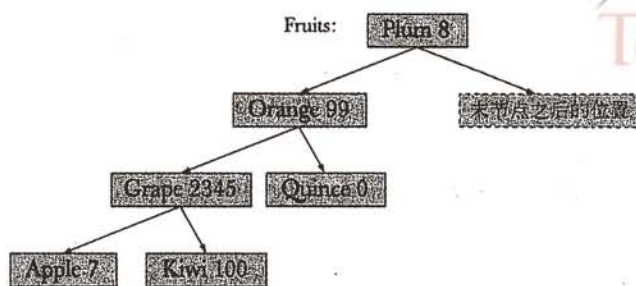
也就是说，对于每个节点，

- 它的左子节点的关键字小于节点的关键字，并且
- 节点的关键字小于它的右子节点的关键字

你可以对书中的每个节点验证这个规则。这就允许我们从树根查找下去。非常奇怪的是，在计算机科学文献中，树是从它的根向下生长的。在这个例子中，根节点是 (Orange, 99)。我们通过比较的方式向下查找，直到找到我们要找的值或它所在的位置。当一棵树的每棵子树的节点数与到根距离相等的所有其他子树的节点数大致相等时，那么这棵树被称为平衡的（就像上面的例子中那样）。平衡树可以减少到达每个节点所经过的节点数。

一个节点可能会保存更多的数据，映射可以用它来保持树中节点的平衡。当一棵树中的每个节点的左、右子树的节点数大致相同时，这棵树就是平衡的。如果一棵有 N 个节点的树是平衡的，我们找到每个节点最多需要查找 $\log_2(N)$ 个节点。这比我们在列表中从开始位置查找一个关键字，平均要查找 N/2 个节点的情况（线性查找时最坏的情况是查找 N 个节点）要好得多。（见

21.6.4 节) 例如, 我们看看一个非平衡的树:



这棵树仍然遵守每个节点的关键字大于它的左子节点、小于它的右子节点的规则:

left->first<first && first<right->first

但是, 这个版本的树是非平衡的, 因此我们知道经过三“跳”到达 Apple 和 Kiwi, 而在平衡树中只需要经过两“跳”。对于有很多节点的树来说, 这个差别是很明显的, 因此用于实现映射的树是平衡的。

我在使用映射时并不需要理解树。这里只是做个合理的假设, 那就是专业人员了解他们的工具的基础知识。我们需要了解的是由标准库提供的映射的接口。这是一个稍微简化过的版本:

```

template<class Key, class Value, class Cmp = less<Key>> class map {
// ...
typedef pair<Key, Value> value_type; // a map deals in (Key, Value) pairs

typedef sometype1 iterator;          // probably a pointer to a tree node
typedef sometype2 const_iterator;

iterator begin();                    // points to first element
iterator end();                      // points one beyond the last element

Value& operator[](const Key& k);      // subscript with k

iterator find(const Key& k);          // is there an entry for k?

void erase(iterator p);               // remove element pointed to by p
pair<iterator, bool> insert(const value_type&); // insert a (key,value) pair
// ...
};
  
```

你可以在 `<map>` 中找到真实的版本。你可以将迭代器想象成一个 `Node*`, 但是你不能将自己的实现依赖于用特定类型来实现迭代器。

`vector` 和 `list` (见 20.5 节和附录 B.4) 的接口的相似性是明显的。最大的不同在于迭代时元素是成对的, 其类型为 `pair<Key, Value>`。这个类型是另一个有用的 STL 类型:

```

template<class T1, class T2> struct pair {
    typedef T1 first_type;
    typedef T2 second_type;
    T1 first;
    T2 second;

    pair():first(T1()), second(T2()) {}
    pair(const T1& x, const T2& y):first(x), second(y) {}
    template<class U, class V>
        pair(const pair<U, V>& p):first(p.first), second(p.second) {}
};

template<class T1, class T2>
pair<T1, T2> make_pair(T1 x, T2 y)
  
```

```
{
    return pair<T1,T2>(x,y);
}
```

我们从标准库中复制 pair 的完整定义及其有用的辅助函数 make_pair()。

注意, 当你对一个映射进行迭代时, 元素将按关键字定义的顺序访问。例如, 如果对例子中的水果进行迭代, 我们将得到:

(Apple,7) (Grape,100) (Kiwi,2345) (Orange,99) (Plum,8) (Quince,0)

我们插入水果的顺序与结果并不匹配。

insert() 操作有一个奇怪的返回值, 我们通常在简单的程序中忽略它。它是对 (key, value) 元素的一对迭代, 如果调用 insert() 插入 (key, value) 对, 它返回的 bool 为 true。如果关键字已经在映射中存在, 则插入失败并且 bool 为 false。

注意, 你可以定义映射使用的顺序的含义, 可以通过提供第三个参数 (Cmp 在映射声明中) 来实现。例如:

```
map<string, double, No_case> m;
```

No_case 定义不区分大小写的比较; 见 21.8 节。less < Key > 定义默认的顺序是小于。

21.6.3 另一个 map 实例

为了更好地体会映射的用途, 我们返回 21.5.3 节中的道琼斯的例子。只有当所有的权重出现在 vector 中它们对应的名字的位置, 这段代码才是正确的。它是隐式的并且容易成为隐蔽的错误来源。这里有很多办法可以解决这个问题, 最有吸引力的办法是使权重与它的公司代号放在一起, 例如 (“AA”, 2.4808)。“代号”是公司名称的缩写, 它用于那些需要简洁表示的情况。类似地, 我们可以将公司代号与它的股票价格放在一起, 例如 (“AA”, 34.69)。最后, 对于那些无法定期访问美国股票市场的人, 我们可以将公司代号与公司名称放在一起, 例如 (“AA”, “Alcoa Inc.”)。也就是说, 我们可以保持三个相关值的映射。

首先, 我们实现 (symbol, price) 映射:

```
map<string,double> dow_price;
// Dow Jones Industrial index (symbol,price);
// for up-to-date quotes see www.djindexes.com
dow_price["MMM"] = 81.86;
dow_price["AA"] = 34.69;
dow_price["MO"] = 54.45;
//...
```

(symbol, weight) 映射如下:

```
map<string,double> dow_weight; // Dow (symbol,weight)

dow_weight.insert(make_pair("MMM", 5.8549));
dow_weight.insert(make_pair("AA", 2.4808));
dow_weight.insert(make_pair("MO", 3.8940));
//...
```

我们使用 insert() 和 make_pair() 函数, 以显示映射中的元素实际是成对的。这个例子也说明了符号的价值; 我们发现标记符号易于读取, 同样重要的是易于书写。

(symbol, name) 映射如下:

```
map<string,string> dow_name; // Dow (symbol,name)
dow_name["MMM"] = "3M Co.";
dow_name["AA"] = "Alcoa Inc.";
dow_name["MO"] = "Altria Group Inc.";
//...
```

通过这些映射，我们可以方便地提炼出各种信息。例如：

```
double alcoa_price = dow_price["AAA"];    // read values from a map
double boeing_price = dow_price["BA"];

if (dow_price.find("INTC") != dow_price.end()) // find an entry in a map
    cout << "Intel is in the Dow\n";
```

通过映射来迭代是容易的。我们只需记住关键字称为 first，而值称为 second：

```
typedef map<string,double>::const_iterator Dow_iterator;

// write price for each company in the Dow index:
for (Dow_iterator p = dow_price.begin(); p!=dow_price.end(); ++p) {
    const string& symbol = p->first;    // the "ticker" symbol
    cout << symbol << '\t'
        << p->second << '\t'
        << dow_name[symbol] << '\n';
}
```

我们甚至可以直接使用映射来完成某些计算。特别是，我们可以计算出索引，就像我们在 21.5.3 节中所做的那样。我们可以通过各自的映射来提取股票价格和权重并将它们相乘。我们可以容易地编写一个函数，以便对任意两个 `map < string, double >` 完成这个操作：

```
double weighted_value(
    const pair<string,double>& a,
    const pair<string,double>& b
) // extract values and multiply
{
    return a.second * b.second;
}
```

现在，我们将这个函数加入 `inner_product()` 的通用版本，并且得到我们的索引的值：

```
double dji_index =
    inner_product(dow_price.begin(), dow_price.end(), // all companies
        dow_weight.begin(),    // their weights
        0.0,                  // initial value
        plus<double>(),        // add (as usual)
        weighted_value);       // extract values and weights
                                // and multiply
```

为什么有人将这类数据保存在映射中，而不是向量中呢？我们使用映射在不同的值之间建立联系，这是一个常见的原因。另一个原因是映射按关键字定义的顺序来保存它的元素。当我们对上面的 `dow` 进行迭代时，我们按字母的顺序来输出符号；如果我们使用向量，则需要自己进行排序。使用映射的最常见的原因是基于关键字查找值时操作简单。对于一个大的序列来说，使用 `find()` 来查找某些东西的速度，比在排序的结构（例如映射）中查找要慢得多。

试一试 运行这个小例子。然后，添加几个你自己选择的公司，以及你自己选择的权重。

21.6.4 unordered_map

为了在一个向量中找到一个元素，`find()` 需要检验所有的元素，从开始到正确值的元素或结尾。这个平均代价与 `vector(N)` 的长度成比例，我们称这个代价为 $O(N)$ 。

为了在映射中找到一个元素，下标操作需要检验树中的所有元素，从根到正确值的元素或叶子。这个代价平均与树的深度成比例。一棵有 N 个节点的平衡二叉树的最大深度为 $\log_2(N)$ ；代价为 $O(\log_2(N))$ 。 $O(\log_2(N))$ 的代价与 $\log_2(N)$ 成比例，与 $O(N)$ 相比实际上是非常好的：

N	15	128	1 023	16 383
$\log_2(N)$	4	7	10	14

实际的代价取决于我们以多快的速度找到这个值，以及比较和迭代的代价。通常，追踪指针（在映射中查找所做的）比增加一个指针（`find()`在向量中所做的）的代价大。

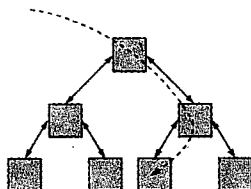
对于有些类型，特别是整数和字符串，我们可以做得比映射的树查找更好。我们不讨论细节，但思路是获得一个关键字，我们计算一个向量中的索引。这个索引称为一个散列值，而使用这种技术的容器通常称为散列表。可能的关键字数量远大于散列表中的存储位置数量。例如，我们经常使用散列函数将数十亿个可能的字符串映射成有 1000 个元素的向量的索引。这可能有些棘手，但是它可以处理得很好并且对实现大的映射特别有用。散列表的主要优点是查找的平均代价接近常数 $O(1)$ ，并且与表中的元素数量无关。很明显，这对于大的映射来说是很大的优点，例如一个有 500 000 个 Web 地址的映射。如果要获得有关散列查找的更多知识，你可以阅读有关 `unordered_map`（在 Web 中）的文档，或者有关数据结构的基础文章（查找散列表和散列）。

我们可以说明在一个（未排序）向量、一个平衡二叉树和一个散列表中如何进行查找，如下图所示：

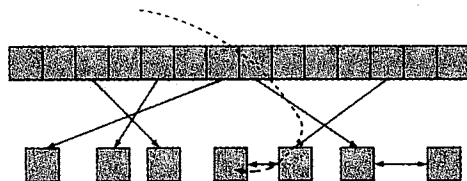
- 在未排序向量中的查找：



- 在映射（平衡二叉树）中的查找：



- 在 `unordered_map`（散列表）中的查找：



STL `unordered_map` 是使用一个散列表来实现的，正如 STL `map` 是使用一个平衡二叉树，而 STL `vector` 是使用一个矩阵来实现的一样。STL 的部分工具用于将所有数据存储和访问方式与通用框架和算法相适应。经验法则是：

- 除非你有好的理由，否则应该使用 `vector`。
- 如果你需要基于值来进行查找（而且你的关键字类型具有合理并且高效的“小于”操作），这时可以使用 `map`。
- 如果你需要在一个大的映射中进行大量查找，并且你不需要有序的遍历（以及是否可以为你关键字类型找到一个好的散列函数），这时可以使用 `unordered_map`。

在这里，我们不会描述 `unordered_map` 的细节。我们可以将 `unordered_map` 与 `string` 或 `int` 类型的关键字共同使用，就像使用 `map` 一样，除非你要迭代的元素是无序的。例如，我们可以重新编写 21.6.3 节中的道琼斯例子，如下所示：

```
unordered_map<string,double> dow_price;
typedef unordered_map<string,double>::const_iterator Dow_iterator;
for (Dow_iterator p = dow_price.begin(); p!=dow_price.end(); ++p) {
```

```

const string& symbol = p->first; // the "ticker" symbol
cout << symbol << '\t'
    << p->second << '\t'
    << dow_name[symbol] << '\n';
}

```

现在，在 `dow` 中查找的速度可能更快。但是，这个变化并不会很显著，这是由于在索引中只有 30 个公司。是否已经保存所有公司在纽约股票交易所的价格，我们可能已经注意到性能上的不同。但是，需要注意一个逻辑上的不同：迭代得到的输出将不会按字母顺序排列。

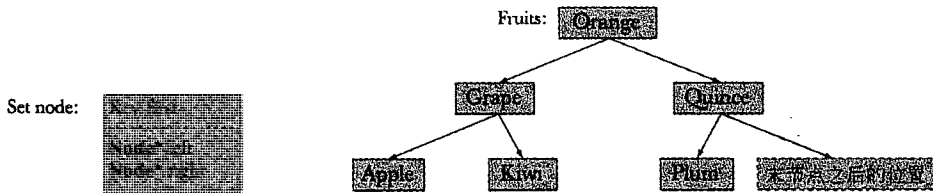
非排序的映射在 C++ 标准文本中是新成员，而不是“第一类成员”，因此说它被定义在技术报告中而不是标准中更恰当。它们被广泛使用，虽然你很少看到它们的前任，某些称为 `hash_map` 的东西。

试一试 使用 `#include <unordered_map>` 来编写一个小程序。如果它没有工作，说明 `unordered_map` 没有被你的 C++ 实现。如果你确实需要 `unordered_map`，你需要下载一个可用的实现（例如，参见 www.boost.org）。

21.6.5 集合

当我们对值没有兴趣时，可以将集合看做一个映射，或看做一个没有值的映射。我们可以用下图表示一个集合：

你可以用集合表示 `map` 例子（见 21.6.2 节）中的水果，如下图所示：



集合什么有用？如果我们看见一个值，这里有很多问题需要我们记住。跟踪可以得到哪种水果（与价格无关）是一个例子；构造一个字典是另一个例子。用做“记录”的集合具有稍微不同的风格；那就是元素是可能包含“大量”信息的对象——我们只需使用一个成员作为关键字。例如：

```

struct Fruit {
    string name;
    int count;
    double unit_price;
    Date last_sale_date;
    // ...
};

struct Fruit_order {
    bool operator()(const Fruit& a, const Fruit& b) const
    {
        return a.name < b.name;
    }
};

set<Fruit, Fruit_order> inventory;

```

接下来，我们看如何使用函数对象来显著地扩大一个 STL 组件适用问题的范围。

由于集合没有一个值类型，因此它也不支持下标（`operator[]()`）。我们必须使用“列表操作”，例如 `insert()` 和 `erase()`。不幸的是，映射和集合都不支持 `push_back()`——原因很明显：集合不是由程序员决定在哪里插入新值，而需要使用 `insert()`。例如：

```
inventory.insert(Fruit("quince",5));
inventory.insert(Fruit("apple", 200, 0.37));
```

集合比映射好的一个优点是你可以直接使用从一个迭代得到的值。由于这里没有 map (见 21.6.3 节) 中的 (key, value) 对, 解引用操作可以得到元素类型的值:

```
typedef set<Fruit>::const_iterator SI;
for (SI p = inventory.begin(), p!=inventory.end(); ++p) cout << *p << '\n';
```

当然, 假设你已经为 Fruit 定义了 <<。

21.7 拷贝操作

在 21.2 节中, 我们认为 find() 是“最简单的有用的算法”。当然, 这一点可以讨论。很多简单的算法是有用的——尽管有些编写起来有些繁琐。当你可以使用其他人编写和调试好的代码时, 为什么要编写新的代码? 当它变得简单和有效时, copy() 将会给 find() 运行上的支持。STL 提供三个版本的拷贝:

拷贝操作	
copy(b,e,b2)	将 [b: e) 拷贝到 [b2: b2 + (e - b))
unique_copy(b,e,b2)	将 [b: e) 拷贝到 [b2: b2 + (e - b)); 抑制临近的拷贝
copy_if(b,e,b2,p)	将 [b: e) 拷贝到 [b2: b2 + (e - b)), 但是仅对满足条件 p 的元素执行

21.7.1 拷贝

基本的拷贝算法的定义如下:

```
template<class In, class Out> Out copy(In first, In last, Out res)
{
    while (first!=last) {
        *res = *first; // copy element
        ++res;
        ++first;
    }
    return res;
}
```

给定一对迭代器, copy() 将它们所指定的序列拷贝到由另一个迭代器指定的序列中 (指向此序列第一个元素)。例如:

```
void f(vector<double>& vd, list<int>& li)
// copy the elements of a list of ints into a vector of doubles
{
    if (vd.size() < li.size()) error("target container too small");
    copy(li.begin(), li.end(), vd.begin());
    // ...
}
```

注意, copy() 的输入序列类型可以与输出序列类型不同。这是对 STL 算法的有效概括: 它们可以用于各种类型的序列, 而无须对实现做不必要的假设。我们需要记住检查是否有足够的空间, 以供输出需要保存其中的元素。程序员的工作是检查空间的大小。STL 算法的目标是最大的通用性和最佳的性能; 它们没有做范围检查和其他保护用户的代价昂贵的测试。有时候, 我们希望由它们来完成, 但是当你想进行检测时, 你可以像我们所做的那样进行检测。

21.7.2 流迭代器

你可能听到过短语“拷贝到输出”和“从输入拷贝”。那是对某种形式的 I/O 的有用的思考, 我们实际上可以像那样使用拷贝。

记住，一个序列是

- 有开始和结尾
- 可以使用 ++ 得到下一个元素
- 可以使用 * 得到当前元素的值

我们可以以这种方式容易地表示输入和输出流。例如：

```
ostream_iterator<string> oo(cout); // assigning to *oo is to write to cout
```

```
*oo = "Hello, "; // meaning cout << "Hello, "
++oo;           // "get ready for next output operation"
*oo = "World!\n"; // meaning cout << "World!\n"
```

你可以想象如何来实现它。标准库中提供了一个 `ostream_iterator` 类型；`ostream_iterator <T>` 是一个迭代器，你可以用它写入类型 `T` 的值。

与此类似，标准库中提供了一个 `istream_iterator <T>`，你可以用它读取类型 `T` 的值：

```
istream_iterator<string> ii(cin); // reading *ii is to read a string from cin
```

```
string s1 = *ii; // meaning cin>>s1
++ii;           // "get ready for the next input operation"
string s2 = *ii; // meaning cin>>s2
```

通过 `ostream_iterator` 和 `istream_iterator`，我们可以为自己的 I/O 使用 `copy()`。例如，我们可以实现一个“粗制滥造的”字典，如下所示：

```
int main()
{
    string from, to;
    cin >> from >> to; // get source and target file names

    ifstream is(from.c_str()); // open input stream
    ofstream os(to.c_str()); // open output stream

    istream_iterator<string> ii(is); // make input iterator for stream
    istream_iterator<string> eos; // input sentinel
    ostream_iterator<string> oo(os, "\n"); // make output iterator for stream

    vector<string> b(ii, eos); // b is a vector initialized from input
    sort(b.begin(), b.end()); // sort the buffer
    copy(b.begin(), b.end(), oo); // copy buffer to output
}
```

迭代器 `eos` 是流迭代器中对“输入结束”的表示。当一个 `istream` 到达输入结束（经常被表示为 `eof`），它的流迭代器相当于默认的流迭代器（这里称为 `eos`）。

注意，我们使用一对迭代器来初始化向量。由于是对一个容器的初始化，一对迭代器 `(a, b)` 表示“将序列 `[a: b)` 读取到容器”。显然，我们使用的一对迭代器是 `(ii, eos)`——输入的开始与结束。这使我们不需要使用 `>>` 和 `push_back()`。我们强烈建议不需要选择

```
vector<string> b(max_size); // don't guess about the amount of input!
copy(ii, eos, b.begin());
```

那些试图猜测输入的绝大多数人，通常会发现他们低估了输入规模，并遇到了严重的问题，对于他们和他们的用户，那就是从结果缓冲区中溢出。这种溢出也是安全问题的一种来源。

试一试 首先，编写程序并用一个小的文件来测试，这个文件中包含几百个单词。

然后，尝试一下强烈不推荐的版本，猜测输入的数量并看当输入缓冲区 `b` 溢出时会发生什么。注意，最坏的情况是在具体的例子中，溢出没有导致任何错误，这样你可以将它

推荐给用户。

在我们的小程序中，我们读取单词然后进行排序。这在当时看来是一个明显的方式，但是我们为什么要将单词放在“错误的位置”，以至于随后我们不得不进行排序？更糟糕的是，我们发现由于一个单词出现在输入中，我们会多次保存和打印这个单词。

我们可以通过用 `unique_copy()` 代替 `copy()` 来解决后一个问题。`unique_copy()` 不会重复拷贝相同的值。例如，如果使用 `copy()`，程序输入

the man bit the dog

并且生成

```
bit
dog
man
the
the
```

如果我们使用 `unique_copy()`，程序将会输出

```
bit
dog
man
the
```

这些新行从哪里来？带有分隔符的输出是很常见的，`ostream_iterator` 的构造函数允许你（可选的）指定在每个值之后打印一个字符串：

```
ostream_iterator<string> oo(os, "\n"); // make output iterator for stream
```

很明显，输出一个新行是适于人类阅读的流行选择，但是我们可能愿意使用空格作为分隔符，我们可以这样写

```
ostream_iterator<string> oo(os, " "); // make output iterator for stream
```

这时将会得到输出

```
bit dog man the
```

21.7.3 使用集合保持顺序

这里有一个更容易的方式来得到输出；那就是使用集合而不是使用向量：

```
int main()
{
    string from, to;
    cin >> from >> to; // get source and target file names

    ifstream is(from.c_str()); // make input stream
    ofstream os(to.c_str()); // make output stream

    istream_iterator<string> ii(is); // make input iterator for stream
    istream_iterator<string> eos; // input sentinel
    ostream_iterator<string> oo(os, " "); // make output iterator for stream

    set<string> b(ii, eos); // b is a set initialized from input
    copy(b.begin(), b.end(), oo); // copy buffer to output
}
```

当我们将值插入一个集合时，重复的值被忽略掉。另外，集合中的元素是保持顺序排列的，因此不需要进行排序。通过使用正确的工具，大多数任务容易完成。

21.7.4 `copy_if`

`copy()` 算法会无条件完成拷贝。`unique_copy()` 算法会限制值相同的相邻元素。第三种拷贝

算法只拷贝满足条件的元素：

```
template<class In, class Out, class Pred>
Out copy_if(In first, In last, Out res, Pred p)
    // copy elements that fulfill the predicate
{
    while (first!=last) {
        if (p(*first)) *res++ = *first;
        ++first;
    }
    return res;
}
```

使用 21.4 节中的 `Larger_than` 函数对象，我们可以找到一个序列中大于 6 的所有元素，如下所示：

```
void f(const vector<int>& v)
    // copy all elements with a value larger than 6
{
    vector<int> v2(v.size());
    copy_if(v.begin(), v.end(), v2.begin(), Larger_than(6));
    // ...
}
```

由于编程时出现的一个错误，这个算法错过了 1998 年的 ISO 标准。这个错误现在已经修改了，但是你仍然可以找到没有 `copy_if` 的实现。如果是这样，请使用本节中的定义。

21.8 排序和搜索

我们经常希望自己的数据是有序的。我们可以通过使用一个数据结构，例如映射或集合，或通过排序来保持顺序。在 STL 中，最常见和有用的排序操作是 `sort()`，我们已经使用过几次。在默认情况下，`sort()` 使用 `<` 作为排序规则，但是我们也可以使用自己的规则：

```
template<class Ran> void sort(Ran first, Ran last);
template<class Ran, class Cmp> void sort(Ran first, Ran last, Cmp cmp);
```

作为一个基于用户特定规则排序的例子，我们将介绍如何进行字符串排序（不考虑特例）：

```
struct No_case {    // is lowercase(x)<lowercase(y)?
    bool operator()(const string& x, const string& y) const
    {
        for (int i = 0; i<x.length(); ++i) {
            if (i == y.length()) return false;    // y<x
            char xx = tolower(x[i]);
            char yy = tolower(y[i]);
            if (xx<yy) return true;                // x<y
            if (yy<xx) return false;               // y<x
        }
        if (x.length()==y.length()) return false; // x==y
        return true;    // x<y (fewer characters in x)
    }
};

void sort_and_print(vector<string>& vc)
{
    sort(vc.begin(),vc.end(),No_case());

    for (vector<string>::const_iterator p = vc.begin(); p!=vc.end(); ++p)
        cout << *p << '\n';
}
```

如果一个序列是有序的，我们不需要用 `find()` 从开始位置查找；我们可以利用顺序进行一次二分查找。二分查找的工作如下：

假设我们在查找值 x ；查看中间元素：

- 如果元素的值等于 x ，我们已经找到它！
- 如果元素的值小于 x ，包含值 x 的元素位于右边，因此我们要查找右半部分（在这半部分进行二分查找）。
- 如果元素的值大于 x ，包含值 x 的元素位于左边，因此我们要查找左半部分（在这半部分进行二分查找）。
- 如果我们已经到达最后一个元素（向左或向右），并且没有找到 x ，那么没有等于值 x 的元素。

对于更长的序列，二分查找比 `find()`（线性查找）的速度更快。二分查找的标准库算法是 `search()` 和 `equal_range()`。我们说的“更长”的含义是什么呢？即使序列中只有 10 个元素，也足以体现出 `search()` 与 `find()` 相比的优势。对于一个有 1000 个元素的序列，`search()` 比 `find()` 的速度要快 200 倍；见 21.6.4 节。

`binary_search` 算法有两种变形：

```
template<class Ran, class T>
bool binary_search(Ran first, Ran last, const T& val);
template<class Ran, class T, class Cmp>
bool binary_search(Ran first, Ran last, const T& val, Cmp cmp);
```

这些算法要求和假设输入序列是排序过的。如果没有排序，那么就会发生“有趣的事”（例如无限循环）。`binary_search()` 告诉我们一个值是否存在：

```
void f(vector<string>& vs)    // vs is sorted
{
    if (binary_search(vs.begin(), vs.end(), "starfruit")) {
        // we have a starfruit
    }

    // ...
}
```

因此，如果我们只关心一个值是否在序列中，那么 `binary_search()` 是理想的。如果我们关心找到的元素，我们可以使用 `low_bound()`、`upper_bound()` 或 `equal_range()`（见 23.4 节和附录 B.5.4）。在我们关心找到哪个元素的情况下，原因在于对象通常比关键字包含更多信息，这里很多元素有可能具有相同的关键字，或者我们希望找到符合某种查找规则的元素。

简单练习

在每一步操作之后打印 `vector`。

1. 定义 `struct Item{string name; int iid; double value; /* ... */}`；并定义一个 `vector<item>` 类型的对象 `vi`，且通过一个文件对其进行初始化以使其包含 10 个元素。
2. 根据 `name` 对 `vi` 排序。
3. 根据 `iid` 对 `vi` 排序。
4. 根据 `value` 对 `vi` 排序；按 `value` 的降序打印（即先打印最大的值）。
5. 插入 `Item("horse shoe", 99, 12.34)` 和 `Item("Canon S400", 9988, 499.95)`。
6. 通过 `name` 删除（擦除）`vi` 中的两个 `Item` 元素。
7. 通过 `iid` 删除（擦除）`vi` 中的两个 `Item` 元素。
8. 采用 `list<Item>` 类型而不是 `vector<Item>` 类型重复上述练习。

现在采用 `map` 类型：

1. 定义一个 `map<string, int>` 类型的对象 `msi`。

2. 插入 10 个(名字, 值)对, 例如, `msi["lecture"] = 21`。
3. 通过 `cout` 输出(名字, 值)对, 输出的格式可由读者自行定义。
4. 删除 `msi` 中的(名字, 值)对。
5. 编写一个函数, 该函数能够从 `cin` 中读取值对并将其存入 `msi` 之中。
6. 从输入读入 10 个值对, 并将它们存入 `msi` 中。
7. 将 `msi` 的元素写入 `cout`。
8. 输出 `msi` 中(整型)数值的总和。
9. 定义一个 `map<int, string>` 类型的对象 `mis`。
10. 将 `msi` 中的值存入 `mis`; 也就是说, 如果 `msi` 的元素为(“lecture”, 21), 则 `mis` 应具有元素(21, “lecture”)。
11. 通过 `cout` 输出 `mis` 的元素。

采用 `vector` 类型:

1. 从一个文件中读入一些浮点值(至少 16 个), 并将其存入一个 `vector<double>` 类型的对象 `vd` 之中。
2. 通过 `cout` 输出 `vd`。
3. 定义一个 `vector<int>` 类型的对象 `vi`, 且 `vi` 具有的元素数量与 `vd` 相同; 将 `vd` 的元素拷贝至 `vi` 中。
4. 通过 `cout` 输出(`vd[i]`, `vi[i]`)值对, 且每一行输出一个值对。
5. 输出 `vd` 元素的总和。
6. 输出 `vd` 元素总和与 `vi` 元素总和的差值。
7. 标准库中存在一种称为 `reverse` 的算法, 且该算法以一个序列(由一对迭代器定义)作为参数; 倒转 `vd`, 并通过 `cout` 输出 `vd`。
8. 计算 `vd` 中元素的平均值, 并将结果输出。
9. 定义一个 `vector<double>` 类型的对象 `vd2`, 并将 `vd` 中所有取值低于(小于)平均值的元素拷贝至 `vd2` 中。
10. 对 `vd` 进行排序, 并输出 `vd`。



思考题

1. 有用的 STL 算法的例子有哪些?
2. `find()` 有什么用途? 至少给出 5 个例子。
3. `count_if()` 有什么用途?
4. `sort(b, e)` 的排序标准是什么?
5. STL 算法如何将一个容器作为其输入参数?
6. STL 算法如何将一个容器作为其输出参数?
7. STL 算法通常如何表示“未找到”或“失败”?
8. 什么是函数对象?
9. 函数对象与函数之间有哪些区别?
10. 什么是谓词?
11. `accumulate()` 有什么用途?
12. `inner_product()` 有什么用途?
13. 什么是关联容器? 至少给出 5 个例子。
14. `list` 是一个关联容器吗? 为什么不是?
15. 什么是二叉树的基本排序属性?
16. 对于一棵树而言, 对其进行平衡有什么含义?
17. `map` 的每一个元素占用了多少空间?
18. `Vector` 的每一个元素占用了多少空间?
19. 当一个(有序的)`map` 可用时, 为什么我们还会使用 `unordered_map`?
20. `set` 如何区别于 `map`?
21. `multi_map` 如何区别于 `map`?

22. 当我们能够“仅仅编写一个简单的循环时”，为什么我们还应使用 `copy()` 算法？
 23. 什么是二分搜索？



术语

<code>accumulate()</code>	<code>find()</code>	搜索	算法
<code>find_if()</code>	序列	应用: ()	函数对象
<code>set</code>	关联容器	一般性	<code>sort()</code>
平衡树	散列函数	排序	<code>binary_search()</code>
<code>inner_product()</code>	流迭代器	<code>copy()</code>	<code>lower_bound()</code>
<code>unique_copy()</code>	<code>copy_if()</code>	<code>map</code>	<code>nordered_map()</code>
<code>equal_range()</code>	谓词	<code>upper_bound()</code>	



习题

- 浏览本章所有内容，并完成所有你未完成的“试一试”练习。
- 找到一个 STL 文档的可靠来源，并列举出所有标准库算法。
- 实现 `count()` 并对其进行测试。
- 实现 `count_if()` 并对其进行测试。
- 如果我们不能通过返回 `end()` 表示“未找到”，那么我们应该怎么办？重新设计并实现 `find()` 和 `count()`，并将迭代器设为指向第一个和最后一个元素。将实现与标准版本进行比较。
- 在 21.6.5 节的水果示例中，我们将 `Fruit` 对象拷贝至 `set` 中。那么，如果我们不希望拷贝 `Fruit` 对象呢？我们可以使用 `set<Fruit*>` 类型的对象作为替代。然而，为了这么做，我们还需要为这一集合定义一个比较操作。通过 `set<Fruit*, Fruit_comparison>` 实现水果示例，并讨论两种实现之间的差别。
- 为 `vector<int>` 类型编写一个二分查找函数（不使用标准函数）。你可以选择任何你喜欢的接口，对该函数进行测试。你是否确信你的二分查找是正确的？现在为 `list<string>` 类型编写一个二分查找函数。对该函数进行测试。这两个二分查找函数彼此之间的相似程度如何？如果你没有学习 STL 的相关知识，你认为这两个二分查找函数的相似程度应该如何？
- 对 21.6.1 节中词频的例子进行修改，以使得它能够根据频率顺序对每一行进行输出（而不是以字典顺序）。一个例子是，输出应该是 3: C++ 而不是 C++: 3。
- 定义一个 `Order` 类，该类包含（顾客）姓名、地址、数据与 `vector<Purchase>` 等成员。`Purchase` 是一个包含（产品）`name`、`unit_price` 和 `count` 等成员的类。定义一种将 `Order` 内容写入文件以及从文件中读入 `Order` 内容的机制。构建一个具有至少 10 个 `Order` 对象所包含内容的文件，将该文件内容读入一个 `vector<Order>` 类型的对象中，根据（顾客）姓名进行排序，并将 `vector<Order>` 中包含的内容写回文件。构建另一个具有至少 10 个 `Order` 对象所包含内容的文件，将文件内容读入一个 `list<Order>` 类型的对象中，根据（顾客）地址进行排序，并将 `list<Order>` 中包含的内容写回文件。通过 `std::merge()` 将两个文件的内容合并，并写入另一个文件。
- 计算上一练习中所形成的两个文件中订单的总价值。一个独立的 `Purchase` 的价值为 `unit_price * count`。
- 设计一个 GUI 接口以向文件输入 `Order` 信息。
- 设计一个 GUI 接口以对包含 `Order` 信息的文件进行查询；例如，“查找 Joe 下的所有订单”，“查询文件 `Hardware` 中订单的总价值”以及“列出文件 `Clothing` 中的所有订单”。提示：首先设计不包含 GUI 接口的程序；然后，在该程序基础上实现 GUI 接口。
- 编写一个程序，该程序能够对文本文件进行“整理”以使所得的文件能够用于一种单词查询程序；也就是说，用空白符取代标点符号，将单词转换为小写形式，用 `do not` 取代 `don't`（等等），以及去除复数形式（例如，`ships` 变为 `ship`）。不要对程序要求太高。例如，确定复数形式通常是困难的，因此如果你同时找到了单词 `ship` 和 `ships`，那么你只需去除 `s`。将程序用于一个真实的至少包含 5000 个单词的文本文件（例如，一篇研究论文）。
- 编写一个程序（使用上一练习的结果），该程序能够回答诸如“文件中单词 `ship` 出现了多少次？”“哪一个

单词出现最频繁?”“文件中最长的单词是什么?”“哪个单词最短?”“列出所有以 s 开头的单词。”“列出所有包含 4 个字符的单词。”

15. 为上一练习中的程序设计一个 GUI 接口。

附言

STL 是 ISO C++ 标准库中关于容器和算法的部分。它提供了非常通用、灵活和有用的基本工具。它能够节省我们的很多工作量:重新发明车轮可能是有趣的,但这并没有什么太大的意义。除非我们有足够的理由不使用 STL,否则我们应该总是使用 STL 容器和基本算法。而且,STL 是泛型编程的一个例子,它展示了实际问题及其解决方案是如何构成一个有用且通用的工具集的。如果你需要对数据进行处理(且大部分程序是这么做的)STL 提供了一个例子、一些思想以及一种有用的方法。

第四部分 拓宽视野

第 22 章 理念和历史

“如果某人说，‘我想要这样一种程序设计语言，我只需说出我希望做什么，它就能帮我完成’，那么就给他一个棒棒糖吧。”

——Alan Perlis

本章简要、有选择性地介绍了程序设计语言的历史和语言的设计理念。这种理念和表达它的语言是达到专业水平的基础。由于本书使用 C++ 语言，因此我们主要关注 C++ 以及影响 C++ 的其他语言。本章旨在对本书所介绍的思想给出其背景和前景。对每种语言，我们会介绍其设计者：一种语言不仅仅是一种抽象的创造，还是一个具体的解决方案——是人对实际中所遇到的问题的回应。

22.1 历史、理念和专业水平

“历史是一堆废话”这是亨利·福特的名言。然而在很久以前，一个相反的观点就被广泛引用了：“不能记住历史的人注定要重复历史。”这里的关键问题是，我们应该选择了解哪一部分历史，又应该摒弃哪一部分：“95% 的事情都是无用的”是另一个相关的论调（虽然我们认为 95% 可能是一个低估的数字）。对于历史和当前实践的关系，我们认为如果对历史没有一定的理解，就不可能达到专业水平。如果你几乎不了解你所在领域的背景，你就很容易被蒙蔽，历史中有太多这样的例子，任何领域都充满了似是而非而又没有实际作用的内容。历史的真正意义在于那些已经在实践中证明自身价值的思想和理念。

我们乐于探讨很多程序设计语言和软件（如操作系统、数据库、图形软件、互联网软件、Web、脚本等）中的关键性思想的起源，你当然还会发现其他很多重要且有用的软件和程序设计领域。我们的篇幅甚至不够（仅仅是）揭开程序设计语言理念和历史的面纱。

程序设计的最终目标一定是生成有用的系统，人们在热烈讨论程序设计技术和语言时，常常会忘记这一点。千万不要忘记它！如果你需要提醒，那么请重新阅读第 1 章。

22.1.1 程序设计语言的目标和哲学

程序设计语言是什么？程序设计语言可以为我们做什么？“程序设计语言是什么”的常见答案包括：

- 指示机器操作的一种工具

- 算法的符号表示法
- 与其他程序员交流的工具
- 进行实验的工具
- 控制电脑设备的一种手段
- 表示各种概念之间关系的一种方法
- 表达高层设计的一种方法

而我们的答案是“以上答案都对，而且还有其他的答案”！显然，我们首先要考虑那些应用于常见领域的程序语言，这是贯穿本章的内容。此外，还有一些专用程序语言和应用于特定领域的程序语言，这些程序语言应用面比较窄并且有着更明确的使用目的。

我们希望程序设计语言具有哪些特性呢？

- 可移植性
- 类型安全
- 定义准确
- 高性能
- 简明表达思想的能力
- 易调试
- 易测试
- 能访问所有系统资源
- 平台独立性
- 可运行在所有平台上
- 长期稳定性
- 能针对应用领域变化做适当的改变
- 易于学习
- 轻量级
- 支持流行的程序设计模式(例如面向对象程序设计和泛型程序设计)
- 有利于程序分析
- 提供大量工具
- 大规模社群的支持
- 适合初学者(如学生、自学者)学习
- 为专业人员(如建筑工程师)提供全面的工具
- 有大量软件开发工具可选用
- 有大量软件组件(如各种库)可选用
- 被一个开放的软件社群所支持
- 被主要的平台厂商所支持(如微软、IBM 等)

不幸的是，我们不能同时拥有所有这些特性。这非常令人失望，因为客观地说每一种特性都很好：它们都能为程序设计提供帮助，没有提供这些特性的程序语言会给程序员带来额外的工作量和复杂性。我们不能同时拥有这些特性的原因很简单：有一些特性是相互排斥的。例如，你不可能在拥有 100% 的平台独立性的同时，还能访问到系统的所有资源；一个程序可以访问某种资源，但这一资源并不是每个平台都会提供，因此这个程序不可能在所有平台都能运行。与之类

似，我们非常希望一种语言(包括它的工具和库)是轻量级的并且容易学习，但是这样就不可能为所有系统和应用领域都提供全面的支持。

这就是语言设计理念的重要之处。对语言、库、工具以及程序的设计者，这些理念指导他们对技术进行选择 and 取舍。没错，当你编写程序的时候，你就是一个设计者，必然要进行技术的选择和取舍。

22.1.2 编程理念

《The C++ Programming Language》的前言中提到，“C++ 语言是一种通用程序设计语言，它的一个主要设计目的就是让那些认真严肃的程序员也能体验到程序设计的乐趣。”这是什么意思？程序设计不就是生产产品吗？不就是正确性、质量和可维护性吗？不就是上市日期吗？就是对软件工程的支持吗？当然，这些说法都没错，但是我们不能忘记程序员——也就是人。考虑另外一个例子，Don Knuth 说过，“Alto 最好的特性就是它不会在晚上运行得更快”。Alto 是来自 Palo Alto 研究中心(PARC)的一台计算机，是最早的“个人计算机”之一。而当时的主流计算机是与之相对的“分时共享计算机”，在白天会有大量用户竞争访问计算机(因而晚上会运行更快)。

程序设计工具和技术存在的意义是为了让程序员能更好地工作并得到出更好的成果。请不要忘记这一点。那么，什么样的指导方针可以帮助程序员以最小的代价设计出最好的软件呢？本书自始至终都在阐述我们对此的理念，因此本节只是对这些内容做一个总结。

我们希望自己的代码有良好结构的主要原因是，在良好结构下，我们可以不必花费很大力气就能修改程序。结构越好，修改程序、寻找和修正错误、增加新特性、移植到新的体系结构中以及优化性能等工作就更容易。这就是我们所说的“良好”的准确含义。

在本节的剩余部分，我们将

- 重新审视我们尝试达到的目标，也就是我们想从代码中得到什么。
- 提出两种一般性的软件开发方法，并说明两者的结合使用比单独使用其中任何一种方法都要更好。
- 思考用代码表达程序结构的关键问题：
 - 直接表达思想
 - 抽象层次
 - 模块化
 - 一致性和最小化

理念就是要拿来用的。它是思考的工具，而不仅仅是用来取悦管理人员和考核人员的华丽的词汇。我们编写的程序应该尽力接近设计理念。当我们陷入程序泥潭的时候，最好回过头来看看，问题是否出在违背了设计理念，有时这是很有帮助的。当评估一个程序时(最好是在交付用户之前)，我们应该寻找那些违背设计理念的部分，这些部分是将来最有可能出问题的地方。应该尽可能广泛地应用设计理念，但也要考虑实践相关问题(例如性能和简单性)和语言的弱点(不存在完美的语言)，这些因素会阻碍我们得到更接近设计理念的结果。

设计理念可以指导我们做出具体的技术决策。例如，我们不能孤立地对一个库的每个接口都做出决策(参见 14.1 节)，这样得到的结果将非常糟糕。正确的方法是：回到我们的基本原则，首先确定对于这个特定的库来说什么是最重要的，然后设计一套一致的接口集合。理想情况下，我们应该在文档和代码注释中清楚地描述出这个特定设计方案所遵循的设计原则及其中的折衷选择。

在一个项目的开始, 首先应该回顾设计理念, 找出它与待解决问题及其解决方案最初思路的相关之处。这是获得和优化设计思路的好方法。随后在设计和开发过程中, 当你陷入困境时, 回过头来查看一下程序中哪个部分偏离设计理念最远——这些就是最有可能隐藏错误、出现设计缺陷的地方。与“在相同的地方反复查看、用相同技术反复寻找错误”的基本调试技术相比, 这种方法提供了另一种调试途径。“错误总是存在于你没有查看的地方——否则你早就找到它了”。

22.1.2.1 我们需要的是什么

典型情况下, 我们需要

- 正确性: 是的, 定义什么是“正确的”非常困难, 但这却是完成工作的重要一步。通常, 对于一个给定项目, 别人会为我们给出正确性的定义, 但是接下来我们还是要理解其含义。
- 可维护性: 每个成功的程序都会随着时间的推移而修改; 它可能会被移植到新的硬件或软件平台上, 可能添加一些新的功能, 或者需要修改新发现的错误。下面一节关于程序结构理念的内容就讨论了可维护性。
- 性能: 性能(“效率”)是一个相对的概念。性能必须与程序的用途相适应。有一种常见的观点: 高效的代码必然是低层的, 结构良好的高层代码会导致低效。而我们的经验恰恰相反, 达到满意性能的途径通常是遵循我们所推荐的理念和方法。例如, STL 就是一个同时兼顾抽象和高效的代码。执迷于低层细节与不屑于低层细节一样容易导致糟糕的性能。
- 按时交付: 交付给用户一个完美的程序, 但时间上却延期了一年, 这一般是无法接受的。显然, 人们的期望常常不切实际, 但是, 我们必须在合理的时间内交付高质量的软件。一种观点认为“按时完成”就意味着粗制滥造, 这并不是事实。相反, 我们发现重视良好的结构(例如, 资源管理、不变式和接口设计)、设计时考虑测试、恰当地使用库(经常是为特定应用或特定领域而设计的库)是如期完工的有效方法。

这些目标要求我们关注代码结构:

- 如果程序中有错误(每一个大型程序都有错误), 清晰的结构有助于发现错误。
- 如果需要让初学者理解程序或者需要修改程序, 清晰的结构会比一大堆细节更容易理解。
- 如果程序遇到了性能问题, 高层程序(更接近设计理念, 并有良好的结构)比低层程序或凌乱的程序更易于性能调整。首先, 高层程序更易于理解。其次, 相对于低层程序, 高层程序在设计早期就已经考虑测试和性能调整因素了。

请注意程序的可理解性。任何能帮助我们理解、分析程序的方法都是有益的。基本上, 规律性总比不规律要好, 只要这种规律性不是因为过度简化而形成的。

22.1.2.2 一般性的方法

编写正确的软件, 有两种方法:

- 自底向上(bottom-up): 只用已证明正确性的组件来构建系统。
- 自顶向下(top-down): 用可能包含错误, 但是能捕获所有错误的组件来构建系统。

有趣的是, 大部分可靠系统都是组合这两种截然相反的方法来构造的。原因很简单: 对于一个大型的真实系统而言, 任何一种方法都无法提供所需的正确性、适应性和可维护性:

- 我们无法构造并“证明”足够多的基本组件来消除所有错误源。
- 当组合有错误的基本组件(如库、子系统、类层次等)来构建最终系统时, 我们无法完全弥补组件的缺陷。

两种方法的结合比任何一种方法单独使用都要好: 我们可以实现(或借用或购买)足够好的组件,

其遗留的错误可以通过错误处理机制和系统化的测试来弥补。而且,如果我们坚持构造更好的组件,就可以用它们构造出更大的组件,从而减少对“凌乱的专用代码”的需求。

测试是软件开发的重要一环,我们将在第 26 章中详细介绍。测试是一种系统化地寻找错误的方法。“尽早测试和日常化测试”是一个流行的观点。我们在程序设计时就应考虑测试问题,努力使测试更简单,并使错误在杂乱的代码中更难以“藏身”。

22.1.2.3 思想的直接表达

当我们表达某事物时(不管它是高层的还是低层的)理想的情况是直接代码来表达,而不是用其他辅助方式。这一理念有几种不同形式:

- 用代码直接表达思想。例如,用特殊类型(如 Month 或 Color)表示参数,比一般类型(如 int)更好。
- 用代码独立地表达相互独立的思想。例如,除少数情况外,标准 sort() 算法可以对任意元素类型的标准容器进行排序;排序、比较操作、容器和元素类型的概念是独立的。而假如我们构造了一个“vector,其对象在自由空间上分配,元素类型是 Object 的派生类,此类定义了一个 before() 成员函数,供 vector::sort() 使用”,那么就得到了一个远不如标准 sort() 那么通用的 sort(), 因为我们对存储、类层次、可用的成员函数、次序等等做出了假设。
- 用代码直接表达思想之间的关系。最常见的可以直接表达的关系是继承(例如, Circle 是一种 Shape)和参数化(例如, vector<T> 表示所有向量都具有的共性,与特定的元素类型无关)。
- 自由组合代码表达的思想——当且仅当这种组合有意义时。例如, sort() 允许我们使用各种不同的元素类型和各种容器,但元素必须支持 < (如果不支持,我们在使用 sort() 时就要用一个额外的参数指定比较操作),且容器必须支持随机访问迭代器。
- 简单地表达简单的思想。遵循上述理念,会导致过度通用的代码。例如,我们可能得出超出任何人需求的过于复杂的类层次(继承结构),或者每个(明显)简单的类都设置了 7 个参数。为了避免每个用户都不得不面对每种可能的复杂情况,我们应尽力提供处理最普遍或者最重要的情形的简单版本。例如,除了使用 op 的通用排序函数 sort(b, e, op) 外,我们还提供隐含使用“<”做为比较操作的版本 sort(b, e)。如果可能的话,我们还想提供使用“<”对标准容器进行排序的 sort(c) 和使用 op 对标准容器进行排序的 sort(c, op) (C++0x 就提供了这两个版本,参见 22.2.8 节)。

22.1.2.4 抽象层

我们更愿意在尽可能高的抽象层上工作,即我们的理念就是以尽可能一般化的方式来表达我们的解决方案。

例如,考虑如何表达电话本的条目(就像我们在 PDA 或手机上保存它那样)。我们可以用 vector<pair<string, Value_type>> 来表达(姓名, 值)对的集合。然而,如果我们实际上总是用姓名来访问该集合的话, map<string, Value_type> 是一种更高层的抽象,它可以使我们免去编写(和调试)访问函数的麻烦。另一方面, vector<pair<string, Value_type>> 又比使用两个数组 srting[max] 和 Value_type[max] 的表示方式抽象程度更高。因为在两个数组的表示方式中,字符串和它的值的关系是隐含的。最低层次的抽象可能是一个 int(元素的个数)加上两个 void*(指向某种程序员了解却不为编译器所知的表示形式)。在我们的例子中,到目前为止介绍的每种表示方式都可认为是非常低层的,因为它们更关注值对的表示形式,而不是其功能。为了更为接近实际应

用,我们可以定义一个直接反映使用方式的类。例如,我们可以设计一个 Phonebook 类,其接口方便使用,然后以它为基础来编写程序。这个 Phonebook 类可以用上述任何一种表示方法来实现。

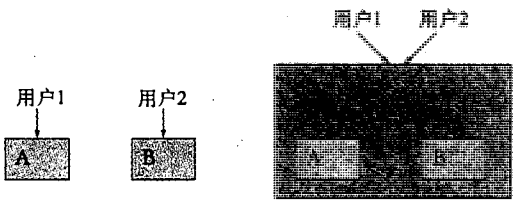
我们更喜欢较高层次的抽象(如果我们有一个适合的抽象机制,并且我们的语言支持这种抽象机制效率较高的话)的原因是,比起在计算机硬件层次表达的解决方案,它更接近于我们思考问题和解决问题的方式。

对于低层抽象,人们使用它的理由往往是“效率”。但要注意,你应该仅在真正需要提高效率时才使用低层抽象(参见 25.2.2 节)。而且,使用低级(更原始的)语言特性不一定能获得更好的性能。相反,有时还会失去进行优化的机会,而高层程序设计则可提供优化可能。例如,使用 Phonebook 类,实现方式可以在 `string[max]` 加上 `Value_type[max]` 和 `map < string, Value_type >` 之间进行选择。对于有些应用,前者更有效,而对另一些应用则是后者更有效。当然,如果应用程序仅包含你的个人通讯录,性能不是主要考虑因素。然而,当我们必须记录和处理数百万个条目的时候,这种权衡就变得很有意义了。更重要的是,如果使用低层特性,一段时间后,处理低层特性就会占用程序员绝大部分时间,以至于没有时间对程序进行改进(在性能或其他方面)。

22.1.2.5 模块化

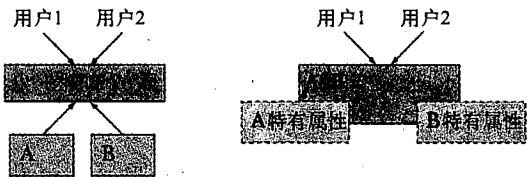
模块化是一种理想。我们希望能用“组件”(如函数、类、类层次、库等)来构建我们的系统,这些组件可以独立构造、理解和测试。理想情况下,我们也希望每个组件都可以用于很多程序中(“重用”)。所谓重用(reuse),就是用以前测试过并且在其他地方已经使用过的组件构建系统,也包括组件的设计和使用等工作。在前面讨论类、类层次、接口设计和泛型程序设计时,我们已经接触过重用的概念。我们所讨论的大部分“程序设计风格”(参见 22.1.3 节)都与设计、实现和使用潜在的“可重用”组件有关。请注意,并不是每个组件都能用于很多程序;一些代码过于专用,难以改进以用于其他地方。

代码中的模块化应该能反映出应用中重要的逻辑差异。我们不是简单地把两个完全无关的类 A 和 B 放在一个“可重用组件”C 中来“提高重用性”。由于需将 A 和 B 的接口合并为 C 的接口,这会使代码复杂化:



如上图所示,用户 1 和用户 2 都使用 C。除非你查看了 C 的内部,否则你可能认为两个用户从组件共享中受益了。从共享(“重用”)中获得的益处(在本例中,实际并未受益)应该包括更好的测试、更少的代码总量、更大的用户基础等。不幸的是,虽然本例有一些过度简化,但所呈现的问题并不是一个特别罕见的现象。

如何做才能解决这个问题呢?也许应该提供一个 A 和 B 公共接口:



两个图旨在表示继承和参数化。在这两种情况下,为了使重用更有价值,所提供的接口必须要比 A 和 B 的接口的简单合并更小。换句话说, A 和 B 必须要有有一个能使用户受益的最小的基本共性集合。请注意,我们又回到了接口问题(参见 9.7 节和 25.4.2 节)和不变式问题(参见 9.4.3 节)。

22.1.2.6 一致性和简约主义

一致性和简约主义是最基本的设计理念。所以我们可能因表象而忽略它们。不过,如果一个设计已经非常杂乱,想要优雅地重新表达它确实很困难。因此,一致性和简约主义应该作为设计标准,在设计过程中就应该遵循,并应该影响到哪怕最微小的程序细节:

- 如果你怀疑一个特性的效用,那么不要添加这个特性。
- 为相似的特性设计相似的接口(和名字),但有一个前提——这种相似性是根本性的。
- 为不同的特性设计不同的名字(或许接口风格也应不同),但前提是这种差异性是本性的。

一致的命名方式、接口风格和实现风格都对维护工作有帮助。当代码一致时,新程序员不需要对庞大系统的每个部分都学习一系列新的规范。STL 就是一个例子(参见第 20~21 章和附录 B.4~B.6)。如果不可能实现一致性(例如,程序包含古老的代码或其他语言编写的代码),一种解决方法是为这些代码设计一个与程序其他部分风格相吻合的接口。与之相对的是,不做任何特殊处理,让外来的(“陌生的”、“糟糕的”)风格直接影响到程序中要访问这些令人厌烦的代码的每个部分。

一种保持简约主义和一致性的方法是:仔细地(并且一贯地)为每个接口做好文档。这样,我们就有更大机会发现不一致的地方和重复的内容。做好前置条件、后置条件和不变式的文档,与仔细留意资源管理和错误报告一样,都是非常有用的。一致的错误处理和资源管理策略,对实现简洁的程序是非常必要的(参见 19.5 节)。

对某些程序员来说,关键的设计原则是 KISS(Keep It Simple, Stupid, 简单的才是最好的)。我们甚至听到有人声称 KISS 是唯一有价值的设计原则。然而,我们更倾向于一些引用不那么广泛的原则,例如“保持事情的简单性”(Keep simple things simple)和“尽可能保持简洁,但不要过分简单化”(Keep it simple, as simple as possible, but no simpler)。后一句话引自阿尔伯特·爱因斯坦,这句话表明,超出了一定界限的过分简化是危险的,因而对设计是有害的。一个显然的疑问是:“为谁简化,和谁比较?”

22.1.3 风格/范型

当我们设计并实现一个程序时,应该保持统一的风格。C++ 支持 4 种基本的风格:

- 过程式程序设计
- 数据抽象
- 面向对象程序设计
- 泛型程序设计

这些风格有时(某种程度上有些自夸)称做“程序设计范型”。除此之外,还有很多其他“范型”,如函数式程序设计、逻辑程序设计、基于规则的程序设计、基于约束的程序设计以及面向方面的程序设计。但 C++ 并不直接支持这些风格,而我们也无法在一本入门书籍中涵盖所有这些内容,所以可以将其留作将来的工作。我们介绍的几种范型/风格也有大量细节不得不略去,也都留作将来进一步地学习:

- 过程式程序设计(procedural programming): 一种利用函数(对参数进行操作)构造程序的思

想。一些数学库函数的例子，和 `sqrt()` 和 `cos()`。C++ 通过函数的概念来支持这种风格（参见第 8 章）。这种风格最有价值的地方在于可以选择多种方式传递参数：传值、传引用或者常量引用。在这种程序设计风格中，数据通常被组织为数据结构（`struct`），而不使用显式的抽象机制（如类的私有数据成员或成员函数）。注意，这种程序设计风格以及函数都是其他风格不可或缺的一部分。

- 数据抽象（data abstraction）：其思想是：首先为应用领域提供一组适合的数据类型，然后使用这些数据类型编写程序。矩阵就是一个经典的例子（参见 24.3 ~ 24.6 节）。这种风格非常注重显式数据隐藏（如使用类的私有数据成员）。标准的 `string` 和 `vector` 都是典型的例子，它们显示出了数据抽象和泛型程序设计的参数化之间的紧密联系。这种风格之所以称做“抽象”，是因为我们通过接口来访问数据类型，而不是直接访问其实现。
- 面向对象程序设计（object-oriented programming）：其思想是：将类型组织为层次结构，以便用代码直接表达它们之间的关系。一个经典的例子是第 14 章的 `Shape` 类。如果各类型间有固有的层次关系的话，这种风格显然是很有价值的。但它也有被滥用的趋势，即人们设计类型的层次结构，并不是基于其内在的关系。因此，当你设计派生类的时候，一定要问一下为什么？你想要表达的是什么？在你的问题中，基类/派生类的差异会对你有什么帮助？
- 泛型程序设计（generic programming）：其思想是：对于具体算法，通过添加参数，来描述算法哪些部分可以变化而不必改变其他部分，从而将算法“提升”到更高的抽象层。第 20 章的 `high()` 是一个简单的算法提升的例子。STL 中的 `find()` 和 `sort()` 算法也是体现了泛型程序设计思想的经典算法。详细情况请参考第 20 ~ 21 章以及下面的例子。

现在把这些风格糅合在一起感受一下！通常人们一提到程序设计风格（“范型”），都是将它们看做毫无关联的：你要么使用泛型程序设计，要么使用面向对象程序设计。但如果你的目标是尽可能好地表达解决方案，就需要组合多种风格了。这里的“好”是指代码易读、易编写、易于维护以及足够高效。考虑这个源于 Simula（参见 22.2.6 节）的经典的“`Shape` 例子”，它通常被看做是面向对象程序设计的例子。第一个解决方案可能是这样的：

```
void draw_all(vector<Shape*>& v)
{
    for(int i = 0; i<v.size(); ++i) v[i]->draw();
}
```

它看起来的确是“当然的面向对象程序设计”。它主要依赖类的层次和虚函数调用为每个给定的 `Shape` 找到正确的 `draw()` 函数；即对一个 `Circle`，它调用的是 `Circle::draw()`，而对于 `Open_polyline`，它调用的是 `Open_polyline::draw()`。但 `vector<Shape*>` 本质上是一个泛型程序设计结构：它依赖于编译时解析的参数（元素类型）。为了强调这一点，我们再来看一个例子，用简单的标准库算法来重写上面的循环：

```
void draw_all(vector<Shape*>& v)
{
    for_each(v.begin(), v.end(), mem_fun(&Shape::draw));
}
```

`for_each()` 的第三个参数是一个函数，`for_each()` 会对序列（由前两个参数指出，参见附录 B.5.1）中每个元素调用该函数。现在，第三个函数调用被假定为一个使用 `f(x)` 语法调用的普通函数（或是一个函数对象），而不是一个使用 `p->f()` 语法调用的成员函数。因此，我们使用标准库函数 `mem_fun()`（参见附录 B.6.2）表明我们实际是希望调用一个成员函数（虚函数 `Shape::draw()`）。

这里的关键点在于 `for_each()` 和 `mem_fun()` 实际上都是模板，一点也不“面向对象”；它们明显属于我们通常所说的泛型程序设计。这里更为有趣的是，`mem_fun()` 是一个返回类对象的独立（模板）函数。换句话说，它也可以轻易地被归为普通的数据抽象风格（非继承性的）甚至是过程化程序设计风格（非数据隐藏）。所以，我们可以说这行代码使用了 C++ 支持的所有 4 种基本风格的主要特点。

但为什么要编写第二个版本的“draw all Shapes”呢？它的功能与第一个版本基本相同，但代码反而更长一些！我们可以给出的一个理由是：与 `for` 相比，用 `for_each()` 表达循环“更显然而且更不容易出错”。但对于大多数人来讲，这并不那么有说服力。还有一种更好的理由：“`for_each()` 表示的是要做什么（遍历序列），而不是怎样去做”。但是，对大多数开发者来说，“有用”才更具说服力：第二个版本为我们指出了一种可以用来解决更多问题的通用方法（用最好的泛型程序设计传统方法）。为什么用 `vector` 而不是 `list` 或一般的序列来保存形状呢？因此，我们可以给出第三个版本（也是更一般的版本）：

```
template<class Iter> void draw_all(Iter b, Iter e)
{
    for_each(b,e,mem_fun(&Shape::draw));
}
```

这个版本适用于所有的形状序列。特别是，它甚至可以用于 `Shape` 数组：

```
Point p(0,100);
Point p2(50,50);
Shape* a[] = { new Circle(p,50), new Triangle(p,p2,Point(25,25)) };
draw_all(a,a+2);
```

由于没有更合适的术语，我们称这种最恰当地混合多种风格的程序设计方式为多范型程序设计（multi-paradigm programming）。

22.2 程序设计语言历史概览

最初的程序设计，就是程序员用手将 0 和 1 刻在石头上！好吧，事情并不是这样的，但也差不了太多。在本节中，我们将从程序设计的（几乎）最初阶段讲起，快速介绍一下程序设计语言发展历史中与 C++ 程序设计相关的一些主要进展。

已有的程序设计语言实在太多了。语言以至少每 10 年 2000 种的速度不断被发明出来，而语言“死亡”的速度也差不多。本节主要介绍过去 60 年中出现的 10 种语言，更多信息请参考 <http://research.ihost.com/hopl/HOPL.html>。在这个网站上，你可以找到三个 ACM SIGPLAN HOPL（程序设计语言历史，History of Programming Languages）会议的全部论文的链接。这些论文都是经过全面的同行评阅的，比一般的网络资源更加完整而可信。本节讨论的语言都是曾在 HOPL 上进行过报告的。注意，如果你在搜索引擎中输入一篇著名论文的完整标题，你有很大机会找到论文的全文。而且，大多数计算机科学家都有自己的主页，你可以在那里找到更多的他们的研究工作的相关信息。

本章对每一种语言的介绍都很简短，实际上每种语言，包括本章未提及的数百种语言，都值得用一整本书来介绍。每一种的语言的内容都经过了精挑细选。我们希望你能够接受这样一个挑战：对每种语言努力学习更多知识，而不是简单地认为“X 语言的全部内容不过如此而已”！请记住，本章介绍的每一种语言都是一个了不起的成就，都曾为我们的世界做出过巨大的贡献。由于篇幅所限，我们无法更全面地介绍这些语言——但总比完全不介绍要好。我们本打算为每一种语言都提供一小段代码，但很遗憾，在本章中并不适合这样做（参见练习 5、6）。

我们见过太多这样的情况：人们在介绍某种人造产品（例如，一种程序设计语言）时，只是简

单地介绍它是什么,或者介绍它是某个无名“开发过程”的产物。这样的介绍歪曲了历史:通常(特别是在形成早期),一种程序设计语言是理念、职业、个人偏好以及外部限制条件作用在一个或(通常是)多个人身上的结果。因此,我们强调与语言相关联的关键人物。并不是 IBM、贝尔实验室、剑桥大学等机构设计了程序语言,而是来自这些机构中的人设计了语言(他们通常与朋友或同事合作完成)。

请注意,有一种奇怪的现象常常扭曲我们对历史的看法。我们为那些著名的科学家或工程师树碑立传之时,都是他们已经功成名就很久之后——已经成为国家科学院院士、皇家学会院士、圣约翰爵士、图灵奖获得者等。换句话说,离他们取得最重要的成就的时间已经过去几十年了。当然,几乎所有著名的科学家和工程师都是在一生中不断地创造出专业成就。但是,当你回过头去审视你所喜欢的程序设计语言和程序设计技巧是如何产生的时候,你可以试着想象一下:一个年轻人(即使是现在,科学和工程领域中的女性仍是太少了,因此假定是一位男性)正在试图计算他是否有足够的钞票请他的女朋友去一个体面的餐厅吃饭;或者是一位父亲正在考虑该将一篇重要的论文提交到哪个会议上,以便这个年轻的家庭能够顺便度个假。至于灰白的胡须、秃顶和过时的服装,那都是很久以后的事情了。

22.2.1 最早的程序语言

从 1949 年开始,当第一代“现代”储存程序式电子计算机出现之时,它们就都具有自己的程序设计语言。那时的每一台计算机都有它自己的语言。当时,算法(例如,行星轨道的计算)的表达和特定机器的指令间是一一对应的。显然,科学家(当时的用户大部分都是科学家)将数学公式记在笔记上,但程序只是一串机器指令的列表。最初的程序列表是十进制或八进制数——与计算机内存中的表示形式完全匹配。后来,汇编器和“自动编码”出现了,即人们发明了用符号名称表示机器指令和机器特性(如寄存器)的语言。这样,程序员可能写出“LD R0 123”就可以将内存地址 123 中的内容读取到 0 号寄存器中。但是,每台机器都有自己的指令集和语言。



如果要选出那个时代有代表性的程序语言设计者,剑桥大学计算机实验室的 David Wheeler 无疑是当然的候选人。1948 年,他编写了运行于储存程序式计算机上的第一个真正的程序(如我们在 4.4.2.1 节提到的“平方表”程序)。大约有 10 个人都声称自己实现了最早的编译器(用于编译机器相关的“自动编码”),David Wheeler 是其中之一,他发明了函数调用(是的,即使是如此显而易见的简单事情,也还是需要某人在某时将它发明出来的)。在 1951 年,他写了一篇杰出的论文来介绍如何设计库,这篇论文的内比那个时代至少超前了 20 年!他与 Maurice Wilkes(在互联网上搜索一下他)和 D. J. Gill 合作完成了第一本关于程序设计的书;他是第一位计算机专业博士学位获得者(1951 年在剑桥大学),后来他的主要贡献在硬件领域(cache 体系结构、早期的局

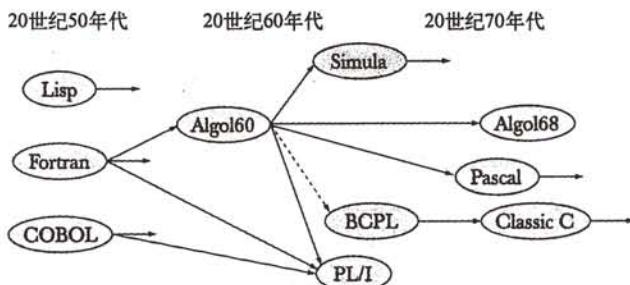
域网)和算法方面(例如, TEA 加密算法(参见 25.5.6 节)和“Burrows-Wheeler 转换”(用于 bzip2 中的压缩算法))。David Wheeler 碰巧还是 Bjarne Stroustrup 的博士论文导师——计算机科学还是一门年轻的学科。David Wheeler 的一些最重要的成就都是在研究生阶段取得的。他后来继续在剑桥大学工作,成为剑桥大学教授和皇家学会院士。

参考文献

- Burrows, M., and David Wheeler. “A Block Sorting Lossless Data Compression Algorithm.” Technical Report 124, Digital Equipment Corporation, 1994.
 Bzip2 link: www.bzip.org/.
 Cambridge Ring website: <http://koo.corpus.cam.ac.uk/projects/earlyatm/cr82>.
 Campbell-Kelly, Martin. “David John Wheeler.” *Biographical Memoirs of Fellows of the Royal Society*, Vol. 52, 2006. (His technical biography.)
 EDSAC: <http://en.wikipedia.org/wiki/EDSAC>.
 Knuth, Donald. *The Art of Computer Programming*. Addison-Wesley, 1968, and many revisions. Look for “David Wheeler” in the index of each volume.
 TEA link: http://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm.
 Wheeler, D. J. “The Use of Sub-routines in Programmes.” Proceedings of the 1952 ACM National Meeting. (That’s the library design paper from 1951.)
 Wilkes, M. V., D. Wheeler, and D. J. Gill. *Preparation of Programs for an Electronic Digital Computer*. Addison-Wesley Press, 1951; 2nd edition, 1957. The first book on programming.

22.2.2 现代程序设计语言的起源

下面是重要的早期程序设计语言的发展历程:



这些程序设计语言的重要性部分是因为它们曾经被广泛使用(目前,在某些情况下仍在广泛使用),另一个原因是,它们是重要的现代程序设计语言的祖先——而且通常还是直接祖先,具有相同的名字。在本节中,我们介绍三种早期程序设计语言——Fortran、COBOL 和 Lisp,大多数现代程序语言的祖先都可以追溯到这三种语言。

22.2.2.1 Fortran

1956 年 Fortran 的发明可能是程序设计语言发展历史中最重要的一步。“Fortran”表示“公式转换”(Formula Translation),其基本思想是将人类(而不是机器)习惯的符号表示转换为高效的机器代码。Fortran 的符号表示法是一种适合于科学家和工程师描述问题数学求解方案的模型,而不是由(最新的)电子计算机所提供的机器指令。

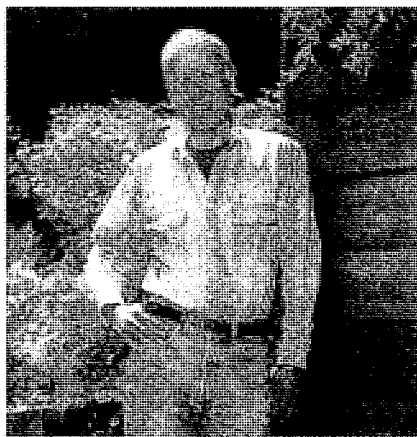
以现代观点来看, Fortran 可以看做对“用代码直接描述应用领域”的首次尝试。它允许程序员像课本中那样书写线性代数公式。Fortran 提供了数组、循环和标准的数学函数(使用标准数学符号,如 $x + y$ 和 $\sin(x)$)。它有一个数学函数的标准库,也提供了 I/O 机制,用户还可以自己定义函数和库。

Fortran 所使用的符号大都是机器无关的,因此 Fortran 代码通常只需很少改动就可以从一台

计算机移植到另一台计算机上。这在当时的发展水平上，是一个巨大的进步。因此，Fortran 被认为是第一个高级程序设计语言。

Fortran 有一个非常重要的优点：由 Fortran 源码生成的机器码可以达到几乎最优的效率。要知道当时的计算机有几个房间那么大，并且极其昂贵（是一个优秀的程序员团队的年薪总和的许多倍），（按现代的标准）它们还慢得出奇（例如 100 000 条指令/秒），内存小得可怜（例如 8K 字节）。不过，人们还是能将有用的程序塞到这些机器中，因此，像 Fortran 这种符号描述方法上的改进，如果不能保持高效率，即便能大大提高程序员的生产率和程序的可移植性，也不会取得成功。

Fortran 在科学与工程计算这一目标领域取得了巨大的成功，并且一直在不断改进、完善。Fortran 语言的主要版本包括 II、IV、77、90、95、03。目前，关于 Fortran77 和 Fortran90 谁应用更广泛的争论仍然在继续。



第一个 Fortran 的定义和实现是由 IBM 的 John Backus 领导的小组完成的：“我们不知道需要什么以及如何做，从某种程度上来说，它就是自然而然地发展起来了。”的确，他又如何能知道呢？之前从没有人做过类似的事情！但是一路走来，他们开发，或者说发明了编译器的基本结构：词法分析、语法分析、语义分析和优化。时至今日，在数值计算优化领域，Fortran 仍然处于领导地位。此外，（在最初的 Fortran 之后）还出现了一种专门用于表示文法的符号系统：Backus-Naur 范式 (BNF)。这种方法在 Algol60（参见 22.2.3.1 节）中首次被使用，现在已经用于大部分现代程序设计语言。在第 6、7 章中，我们也使用了某个版本的 BNF 来描述文法。

很久之后，John Backus 开辟了一个全新的程序设计语言分支（“函数式程序设计”）。与基于读写内存位置的从机器出发的方式相反，这种程序设计风格主张用数学方式来编写程序。需要注意的是，纯数学是没有赋值的概念的，甚至连操作的概念也没有。纯数学只是在一组给定的条件下，“简单地”声明什么肯定是真的。函数式程序设计的思想部分源于 Lisp（参见 22.2.2.3 节），一些函数式程序设计的思想也反映在 STL 中（参见第 21 章）。

参考文献

- Backus, John. “Can Programming Be Liberated from the von Neumann Style?” *Communications of the ACM*, 1977. (His Turing award lecture.)
- Backus, John. “The History of FORTRAN I, II, and III.” *ACM SIGPLAN Notices*, Vol. 13 No. 8, 1978. Special Issue: History of Programming Languages Conference.
- Hutton, Graham. *Programming in Haskell*. Cambridge University Press, 2007. ISBN 0521692695.
- ISO/IEC 1539. *Programming Languages – Fortran*. (The “Fortran 95” standard.)

Paulson, L. C. *ML for the Working Programmer*. Cambridge University Press, 1991. ISBN 0521390222.

22.2.2.2 COBOL

COBOL(面向商业的通用语言, The Common Business-Oriented Language)曾是面向商业应用程序员的主要语言(目前在某些情况下仍然是), 就像 Fortran 曾是面向科学应用程序员的主要语言(目前在某些情况下仍然是)一样。COBOL 主要用于数据处理:

- 数据复制
- 数据存储和检索(如记账)
- 打印(如报表)

计算被看做小事(在 COBOL 的核心应用领域通常是这样的)。人们期望/宣称 COBOL 是如此接近“商务英语”, 连管理人员都很可能用它来编程, 从而很快就会使程序员变得多余。这是那些热衷于削减程序设计开支的经理的良好愿望, 但从来没有实现, 哪怕接近实现。

COBOL 最初是由一个委员会(CODASYL)在 1959 ~ 1960 年设计的, 这个委员会是由美国国防部和一些主要的计算机制造商发起的, 其目的是解决商业计算的需求。COBOL 的设计直接以 Grace Hopper 发明的 FLOW-MATIC 语言为基础。她的贡献之一就是使用了一种与英语十分接近的语法(与之相对的是由 Fortran 开创的使用数学符号的语法, 目前仍然占据主导地位)。与 Fortran 以及其他所有成功的语言一样, COBOL 也历经不断的演化和发展, 其主要的版本包括 60、61、65、68、70、80、90 和 04。

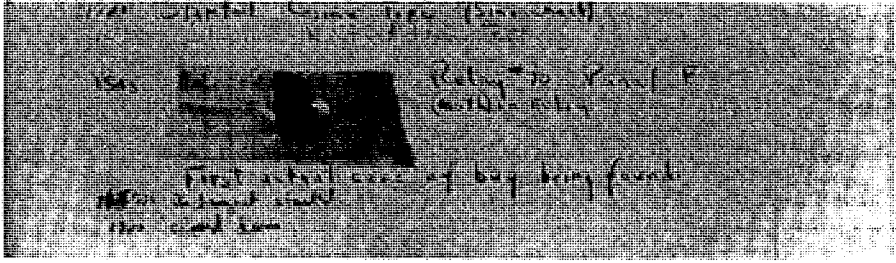
Grace Murray Hopper 拥有耶鲁大学的数学博士学位。在第二次世界大战期间她为美国海军工作, 研究最早期的计算机。在(早期的)计算机工业界工作了几年后, 她又回到了海军。



“海军少将 Grace Murray Hopper 博士(美国海军)在早期的计算机程序设计领域做出了杰出的贡献。她将软件开发思想的研究作为一生的事业, 作为这个领域的领路人, 是她引领了从原始的程序设计技术到使用复杂编译器的转变。她坚信‘我们原来就是这么做的’不是继续这么做的必然理由。”

——Anita Borg 1994 年“Grace Hopper Celebration of Women in Computing”会议上的发言

Grace Murray Hopper 一直被认为是第一个将计算机中的错误称为“bug”的人。她无疑是最早使用这一术语并且在文档中对此进行了记载的人:



我们可以看到，这是一只真正的虫子（一只飞蛾），它直接引起了一个硬件故障。但是现代计算机故障多为软件故障，很少能如此生动地呈现出来。

参考文献

A biography of G. M. Hopper: <http://tergestesoft.com/~eddysworld/hopper.htm>.

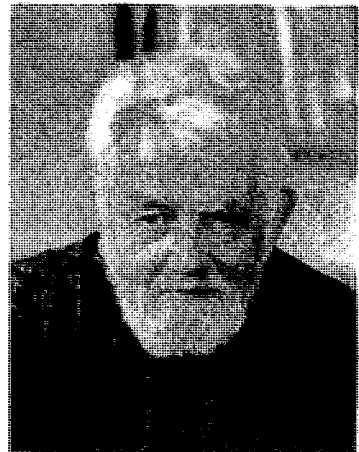
ISO/IEC 1989:2002. *Information Technology - Programming Languages - COBOL*.

Sammet, Jean E. "The Early History of COBOL." *ACM SIGPLAN Notices*, Vol. 13 No. 8, 1978. Special Issue: History of Programming Languages Conference.

22. 2. 2. 3 Lisp

Lisp 最初是 John McCarthy 1958 年在麻省理工学院设计的一种语言，主要用于链表和符号处理（也因此而得名，LISt Processing）。与编译型语言不同，最初的 Lisp 是解释型语言（现在通常也是）。Lisp 有几十种（可能更多，有几百种）方言。实际上，人们经常说“Lisp 默认就是复数”。目前最流行的版本是 Common Lisp 和 Scheme。这种语言曾经是（现在也是）人工智能领域研究的支柱（虽然发布的产品通常是用 C 或者 C++ 实现的）。Lisp 最主要的灵感源泉是 λ 演算（的数学思想）。

在各自的应用领域中，Fortran 和 COBOL 的设计目标都是为了解决现实世界中的问题。而 Lisp 社区则更加关注程序设计本身和程序的优雅性。通常这些努力都很成功。Lisp 是第一种将自身定义与硬件分离的语言，也是第一种将语义建立在某种数学形式之上的语言。如果说 Lisp 有一个特定应用领域的话，也很难给出其准确定义：“人工智能”或者“符号计算”都不像“商业处理”和“科学计算”那样能清楚地对应到某种普通日常工作。在很多现代语言，尤其是函数式语言中都能发现来自 Lisp（或来自 Lisp 社区）的设计理念。



John McCarthy 在加州理工学院获得数学学士学位，在普林斯顿大学获得数学博士学位。你可能已经注意到了，很多程序语言的设计者都来自数学专业。在麻省理工学院完成了他载入史册的

工作后, McCarthy 于 1962 年来到斯坦福大学, 参与建立了斯坦福人工智能实验室。他被公认为人工智能一词的发明人, 并在这一领域做出了许多贡献。

参考文献

- Abelson, Harold, and Gerald J. Sussman. *Structure and Interpretation of Computer Programs, Second Edition*. MIT Press, 1996. ISBN 0262011530.
- ANSI INCITS 226-1994 (formerly ANSI X3.226:1994). *American National Standard for Programming Language-Common LISP*.
- McCarthy, John. "History of LISP." *ACM SIGPLAN Notices*, Vol. 13 No. 8, 1978. Special Issue: History of Programming Languages Conference.
- Steele, Guy L. Jr. *Common Lisp: The Language*. Digital Press, 1990. ISBN 1555580416.
- Steele, Guy L. Jr., and Richard Gabriel. "The Evolution of Lisp." Proceedings of the ACM History of Programming Languages Conference (HOPL-2). *ACM SIGPLAN Notices*, Vol. 28 No. 3, 1993.

22.2.3 Algol 家族

在 20 世纪 50 年代后期, 许多人认为程序设计变得过于复杂、专用、不科学。人们还觉得程序设计语言的种类过于繁多, 而且这些语言的组合既没有充分考虑通用性, 也没有坚实的理论基础。从那时起, 这种质疑的观点多次被提及, 但真正的改变来自 IFIP (国际信息处理联合会, the International Federation of Information Processing) 支持的一个工作组。在短短几年时间内, 他们创立了一种崭新的程序设计语言。这种语言颠覆了我们对于程序设计语言及其定义的认识。多数的现代程序设计语言, 包括 C++, 都曾从中受益。

22.2.3.1 Alogl60

Algol (ALGOrithmic Language) 是由 IFIP 2.1 工作组设计的, 是对现代程序设计语言概念的重要突破:

- 词法作用域
- 使用文法定义语言
- 语法和语义规则明确分离
- 语言定义和实现明确分离
- 系统化地使用(静态, 即编译时)类型
- 直接支持结构化编程

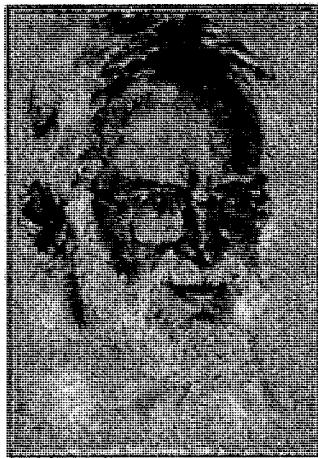
“通用编程语言”的理念就源于 Algol。在它之前, 程序设计语言都是专门服务于科学(如 Fortran)、商业(如 COBOL)、表处理(如 Lisp)、仿真等的。在这些语言中, 与 Algol60 最接近的是 Fortran。

不幸的是, Algol60 的广泛使用从未走出学术领域。因为很多工业界人士认为它“过于古怪”, Fortran 程序员认为它“太慢”, COBOL 程序员认为它“对商业处理的支持不足”, Lisp 程序员认为它“不够灵活”, 大多数工业界人士(包括控制程序设计工具投资的经理)认为它“太学院派”, 很多美国人认为它“太欧洲”。多数的批评是正确的, 例如, Algol60 报告中没有定义任何 I/O 机制! 但是, 同时代的其他语言也存在类似的问题, 不能因此而否定 Algol 语言的重要地位, Algol 为很多领域定下了新的标准。

Algol60 的一个问题是没人知道如何实现它。由 Peter Naur (Algol60 报告的编写者) 和 Edsger Dijkstra 领导的程序员团队解决了这一问题。

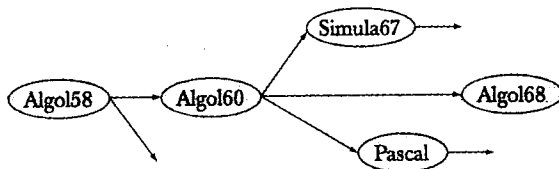
Peter Naur 就读于哥本哈根大学(学习天文), 随后在哥本哈根理工大学(DTH)工作, 并为丹麦计算机制造商 Regnecentralen 工作。他最早接触程序设计是在(1950 ~ 1951 年)在英国剑桥大学计算机实验室(当时丹麦还没有计算机), 之后他在这个领域的杰出贡献跨越了学术界和工业界。

他是 Backus-Naur 范式的共同发明人,也是最早提议对程序进行形式化推理的人(大约在 1971 年, Bjarne Stroustrup 从 Peter Naur 的学术论文中第一次接触到了不变式的使用)。Naur 从未停止对计算科学前景的思考,一直在关注程序设计中人的因素。事实上,他后期的研究工作已经可以归为哲学范畴了(除了他认为传统的学院派哲学毫无意义)。他是哥本哈根大学第一位 Datalogi 教授(datalogi 是丹麦语,最好翻译为“informatics”,信息学;Peter Naur 非常不喜欢“计算机科学”(computer science)一词,认为这是彻底的用词不当,因为他并不认为“计算”指的就是“计算机”)。



Edsger Dijkstra 是另一位史上最伟大的计算机科学家。他在莱顿学习物理,但早期的计算相关的工作是在阿姆斯特丹数学中心进行的。他后来又在很多地方工作过,包括埃因霍温理工大学、宝来公司和得州大学奥斯汀分校。除了 Algol 语言方面的杰出工作外,他还是利用数学逻辑研究程序设计和算法的先驱和积极倡导者,此外还是 THE 操作系统设计者和实现者之一。THE 是最早的具有系统化处理并发操作能力的操作系统之一。THE 表示“Technische Hogeschool Eindhoven”——Edsger Dijkstra 当时工作的大学。他的最著名的论文“Go-To Statement Considered Harmful”,令人信服地阐述了非结构化控制流所存在的问题。

Algol 家族树如下所示:



注意, Simula67 和 Pascal 这两种语言是很多(几乎是所有的)现代语言的祖先。

参考文献

- Dijkstra, Edsger W. "Algol 60 Translation: An Algol 60 Translator for the x1 and Making a Translator for Algol 60." Report MR 35/61. Mathematisch Centrum (Amsterdam), 1961.
- Dijkstra, Edsger. "Go-To Statement Considered Harmful." *Communications of the ACM*, Vol. 11 No. 3, 1968.
- Lindsey, C. H. "The History of Algol68." *Proceedings of the ACM History of Programming Languages Conference (HOPL-2)*. *ACM SIGPLAN Notices*, Vol. 28 No. 3, 1993.
- Naur, Peter, ed. "Revised Report on the Algorithmic Language Algol 60." A/S Regnecentralen (Copenhagen), 1964.

Naur, Peter. "Proof of Algorithms by General Snapshots." *BIT*, Vol. 6, 1966, pp.310-16. Probably the first paper on how to prove programs correct.

Naur, Peter. "The European Side of the Last Phase of the Development of ALGOL 60." *ACM SIGPLAN Notices*, Vol. 13 No. 8, 1978. Special Issue: History of Programming Languages Conference.

Perlis, Alan J. "The American Side of the Development of Algol." *ACM SIGPLAN Notices*, Vol. 13 No. 8, 1978. Special Issue: History of Programming Languages Conference.

van Wijngaarden, A., B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker, eds. *Revised Report on the Algorithmic Language Algol 68* (Sept. 1973). Springer-Verlag, 1976.

22.2.3.2 Pascal

在 Algol 家族树中, Algol68 语言是一个巨大、雄心勃勃的项目。像 Algol60 一样, 它也是由“Algol 委员会”(IFIP 工作组 2.1)负责。但是看上去它“永远”也不能完成, 以至于很多人失去了耐心, 并怀疑这样一个项目所产生的成果是否真的有用。Algol 委员会的一个成员——Niklaus Wirth, 决定设计、实现自己的语言。这就是 Pascal, 它也是源于 Algol, 但与 Algol68 不同, 它是 Algol60 的简化。

Pascal 于 1970 年完成, 它确实很简单, 带来的一个后果就是不够灵活。人们一般认为它只适合教学, 但早期的相关论文都把它描述为 Fortran 的替代品, 用于当时的超级计算机。Pascal 确实非常容易学习, 并且随着一个可移植性极好的版本的实现, 它逐渐成为一种十分流行的教学语言, 但实践证明它没有对 Fortran 造成任何威胁。



Pascal 是瑞士苏黎世联邦理工学院(ETH)的 Niklaus Wirth 教授(上面的照片分别是 1969 年和 2004 年拍摄的)的杰作。他在加州大学伯克利分校获得了电子工程和计算机科学博士学位, 并和加州有着终生的不解之缘。如果说要推选一位程序语言设计终极专家的话, Wirth 教授是这个世界上最配得上这个称号的人。在 25 年时间里, 他设计和实现了下列语言:

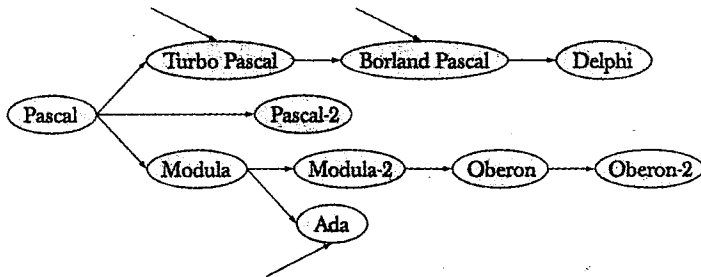
- Algol W
- PL/360
- Euler
- Pascal
- Modula
- Modula-2
- Oberon

- Oberon-2
- Lola(一种硬件描述语言)

Niklaus Wirth 把这些工作当做对简洁性的无止境的追求。他的工作对这个领域产生了巨大的影响。学习这一系列语言是一种非常有趣的练习。Wirth 教授是唯一在 HOPL 会议上提出过两种语言人。

最终,纯粹的 Pascal 被证明对于工业界来说太简单、太严格了。20 世纪 80 年代,主要是在 Anders Hejlsberg 的努力下,将 Pascal 从消亡的边缘拉了回来。Anders Hejlsberg 是 Borland 的三位创始人之一。最初他设计并实现了 Turbo Pascal(与其他实现相比,有着更加灵活参数传递机制),后来又设计了类似 C++ 的对象模型(但是仅有单一继承,并有更好的模块机制)。他就读于哥本哈根理工大学(Peter Naur 曾工作过的地方)——世界有时真的是很小啊。Anders Hejlsberg 后来为 Borland 设计了 Delphi 语言,为微软设计了 C#语言。

Pascal 家族树(经过必要的简化后)如下所示:



参考文献

- Borland/Turbo Pascal. http://en.wikipedia.org/wiki/Turbo_Pascal.
- Hejlsberg, Anders, Scott Wiltamuth, and Peter Golde. *The C# Programming Language, Second Edition*. Microsoft .NET Development Series. ISBN 0321334434.
- Wirth, Niklaus. "The Programming Language Pascal." *Acta Informatica*, Vol. 1 Fasc 1, 1971.
- Wirth, Niklaus. "Design and Implementation of Modula." *Software-Practice and Experience*, Vol. 7 No. 1, 1977.
- Wirth, Niklaus. "Recollections about the Development of Pascal." Proceedings of the ACM History of Programming Languages Conference (HOPL-2). *ACM SIGPLAN Notices*, Vol. 28 No. 3, 1993.
- Wirth, Niklaus. *Modula-2 and Oberon*. Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages (HOPL-III). San Diego, CA, 2007. <http://portal.acm.org/toc.cfm?id=1238844>.

22.2.3.3 Ada

Ada 语言是为美国国防部专门设计的。特别是,它是一种适合于为嵌入式系统编写可靠、可维护程序的语言。它的最明显的祖先是 Pascal 和 Simula(参见 22.2.6 节)。Ada 设计小组的领导人是 Jean Ichbiah——他曾经是 Simula 用户组的主席。Ada 的设计强调:

- 数据抽象(但直到 1995 才引入继承机制)
- 强大的静态类型检查
- 语言直接支持并发性

Ada 的设计目标是希望在程序设计语言中体现软件工程思想。其结果就是,美国国防部设计出的不是一门语言,而是设计语言的一套详细流程。众多的人和组织都对此做出了贡献,整个项目是通过一系列的竞赛来推进的,首先是评选出最佳的语言规范,随后就是设计能体现出最佳规范思想的最佳语言。这个长达 20 多年的巨大项目(1975~1998 年)从 1980 年开始由 AJPO(Ada Joint

Program Office, Ada 联合计划组织)负责管理。

在 1979 年,语言设计完成,并以 Augusta Ada Lovelace 女士(著名诗人拜伦勋爵的女儿)的名字命名。Lovelace 女士被认为是第一位现代程序员(当然,这里“现代”的定义很宽泛),因为她在 19 世纪 40 年代曾和 Charles Babbage(剑桥大学的卢卡斯数学教授——牛顿曾经担任过这一职位)一起工作,研究一种革命性的机械式计算机。不幸的是,Babbage 的机器没能成功地成为一个实用工具。



正是因为其设计遵循了详尽的流程,Ada 被认为是终极的“委员会设计”语言。作为最终赢得胜利的设计团队的领导者,来自法国 Honeywell Bull 公司的 Jean Ichbiah 却强烈地否认这一点。然而,我猜测(基于和他的讨论),如果没有这个详细流程的束缚,他本可以设计出更好的语言。

Ada 被美国国防部确定为军事应用程序强制使用语言已经有很多年历史了,以至于有这样的说法,“Ada 不仅仅是一个好的思想,更是一项法律”!最初,Ada 只是“强制”使用而已,但随着许多项目获得“豁免权”来使用其他语言(通常是 C++),美国国会通过了一项法律,要求大多数军事应用程序必须使用 Ada。后来,这一法律由于所面对的商业和技术上的现实问题,不得不废除了。Bjarne Stroustrup 是极少数工作成果曾被美国国会禁止的人之一。

我们坚信,与它获得的声誉相比,Ada 实际上要好得多。假如美国国防部不那么强硬地推广 Ada,而且能够正确应用它的话(用做开发过程、软件开发工具、文档等的标准),它也许会成功得多。直到现在,Ada 仍是航空应用程序和类似的高级嵌入式系统应用程序的重要编程语言。

Ada 在 1980 年成为军事标准,1983 年成为 ANSI 标准(第一个 Ada 实现在 1983 年完成——在第一个标准完成之后三年),1987 年成为 ISO 标准。这个 ISO 标准在 1995 年被全面地(当然在保证兼容性的前提下)进行了修订。重要的改进包括更灵活的并发机制以及支持继承。

参考文献

Barnes, John. *Programming in Ada 2005*. Addison-Wesley, 2006. ISBN 0321340787.

Consolidated Ada Reference Manual, consisting of the international standard (ISO/IEC 8652:1995). *Information Technology - Programming Languages - Ada*, as up dated by changes from *Technical Corrigendum 1* (ISO/IEC 8652:1995:TC1:2000).

Official Ada homepage: www.usdoj.gov/crt/ada/.

Whitaker, William A. *ADA - The Project: The DoD High Order Language Working Group*. Proceedings of the ACM

History of Programming Languages Conference (HOPL-2). *ACM SIGPLAN Notices*, Vol. 28 No. 3, 1993.

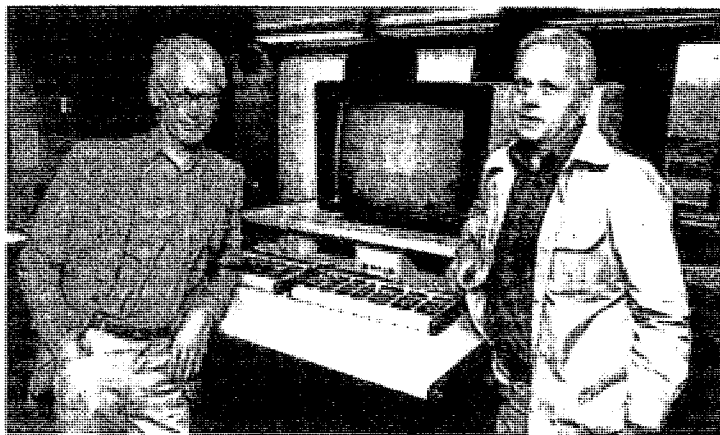
22.2.4 Simula

20 世纪 60 年代中期, Kristen Nygaard 和 Ole-Johan Dahl 设计了 Simula, 当时他们在挪威计算中心和奥斯陆大学工作。Simula 毫无疑问是 Algol 语言家族的成员。实际上, Simula 几乎就是 Algol60 的超集。但是, 我们选择单独对 Simula 进行介绍, 这是因为现在人们所说的“面向对象程序设计”的大多数基本思想都来自 Simula。它是第一种提供继承和虚函数机制的语言。术语类 (class) 表示“用户自定义类型”和虚函数 (virtual function) 表示可以覆盖并可通过基类接口调用的函数, 都来源于 Simula。

Simula 的贡献并不局限于语言方面, 它更重要的贡献是提出了明确的面向对象设计的概念, 这基于用代码来建模现实世界现象的思路:

- 用类和类对象表示思想。
- 用类层次(继承)表示层次关系。

因此, 程序变成一组相互作用的对象, 而不是单个的庞然大物。



Kristen Nygaard 是 Simula 的共同发明人(另一位是 Ole-Johan Dahl, 照片中左侧戴眼镜者), 他在很多方面都堪称巨人(包括身高), 他的热情和慷慨也完全配得上这样的声誉。他构思了面向对象编程和设计(特别是继承)的基本思想, 并在此后的几十年间不断探索这些思想的含义。他从不满足于简单、短期和短视的答案。他还数十年如一日地积极投身到社会活动中。如果挪威不是置身于欧盟之外的话, 他本应获得更高的声望。但他认为欧盟有可能成为一个中央集权和官僚主义的噩梦般的机构, 它不会关心挪威这样的边缘小国的需求。20 世纪 70 年代中期, Kristen Nygaard 花了大量时间在丹麦奥尔胡斯大学的计算机科学系(当时, Bjarne Stroustrup 在那里攻读硕士学位)。

Kristen Nygaard 在奥斯陆大学获得数学硕士学位。他在 2002 年去世, 就在他即将(与他终生的朋友 Ole-Johan Dahl 一起)获得 ACM 图灵奖之前的一个月, 这个奖项对计算机科学家来说是最高的荣誉。

Ole-Johan Dahl 是一位更传统的学者。他的研究兴趣主要集中在语言规范和形式化方法方面。1968 年, 他成为奥斯陆大学第一位信息学(计算机科学)全职教授。

2000 年 8 月, Dahl 和 Nygaard 被挪威国王授予圣奥拉夫高级骑士勋章。在他们的家乡, 纯粹的计算机技术人员才能获得如此高的荣誉!



参考文献

- Birtwistle, G., O.-J. Dahl, B. Myhrhaug, and K. Nygaard: *SIMULA Begin*. Student-litteratur(Lund, Sweden), 1979. ISBN 9144062125.
- Holmevik, J. R. "Compiling SIMULA: A Historical Study of Technological Genesis." *IEEE Annals of the History of Computing*, Vol. 16 No. 4, 1994, pp. 25-37.
- Kristen Nygaard's homepage: <http://heim.ifi.uio.no/~kristen/>.
- Krogdahl, S. "The Birth of Simula." Proceedings of the HiNC 1 Conference in Trondheim, June 2003 (IFIP WG 9.7, in cooperation with IFIP TC 3).
- Nygaard, Kristen, and Ole-Johan Dahl. "The Development of the SIMULA Languages." *ACM SIGPLAN Notices*, Vol. 13 No. 8, 1978. Special Issue: History of Programming Languages Conference.
- SIMULA Standard. *DATA processing - Programming languages - SIMULA*. Swedish Standard, Stockholm, Sweden (1987). ISBN 9171622349.

22.2.5 C

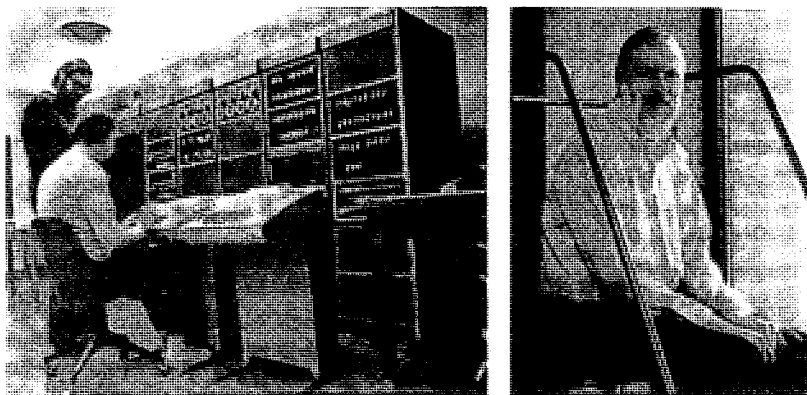
在 1970 年，一件“众所周知”的事情是，重要系统的程序设计——特别是操作系统的实现，必须使用汇编语言，从而不具备可移植性。这与 Fortran 出现之前科学计算程序设计的处境很相像。一些个人和组织开始着手挑战这个成见，最终，C 语言（参见第 27 章）成为这些工作中的最成功者。

Dennis Ritchie 在位于新泽西州茉莉山的贝尔电话实验室计算科学研究中心设计并实现了 C 语言。C 的魅力在于它是一种简单的编程语言，而这种简单性是慎重规划后的结果，而且 C 语言与硬件的基本特性关联非常紧密。目前 C 语言版本的复杂性（其中大多数出于兼容性考虑也出现在了 C++ 中）大多数是在 Dennis Ritchie 的最初设计之后添加进来的，并且有某些情况并不符合他的原意。C 的成功部分是因为很早就被广泛使用，但它真正的强大之处在于语言特性到硬件设施直接映射（参见 25.4 ~ 25.5 节）。Dennis Ritchie 曾简单地将 C 描述为“一种强类型、弱检查的语言”；也就是说，C 是一种静态（编译态）类型的系统，在程序中不按对象定义的方式使用它是非法的——但 C 编译器又不会检查这种问题。但当资源有限，如内存只有 48K 字节时，这样做可以得到更简短的代码，还是有意义的。在 C 投入应用后不久，人们设计了一种称为 lint 的程序，它将类型系统验证从编译器中分离出来。

Dennis Ritchie 与 Ken Thompson 一起发明了 UNIX，它无疑都是最有影响力的操作系统。C 一直都与 UNIX 操作系统紧密联系在一起。近年来，又与 Linux 和开源项目的发展紧密相连。

在为贝尔实验室计算机科学研究中心工作了 40 年之后，Dennis Ritchie 现在已经从朗讯公司

贝尔实验室退休了。他在哈佛大学获得物理学学士学位和应用数学博士学位。



在 1974 ~ 1979 年间, Bell 实验室中的很多人都对 C 的设计和应用产生过影响。Doug McIlroy 是所有人都喜欢的评论家、讨论伙伴和创意丰富的人。他影响了 C、C++、UNIX 和其他很多研究工作。



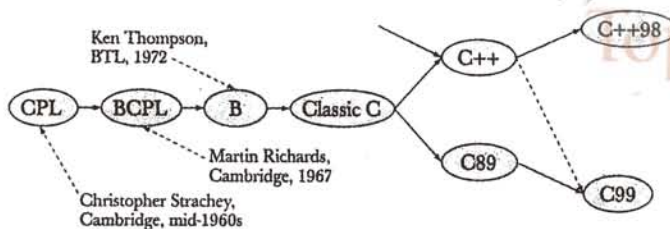
Brian Kernighan 是一位杰出的程序员和作家。他的代码和文章都是清晰性的典范。本书的风格就部分来源于他的杰作《The C Programming Language》(Brian Kernighan 和 Dennis Ritchie 合著, 因而被称为“K&R”)中指南那部分。

仅有好的思想是不够的; 为了能被更大范围的人们所用, 应该将这些思想归约到最简单的形式, 并以目标读者群中更多人能够接受的方式阐述清楚。在表达思想时, 冗长啰嗦是最可怕的敌人, 同样可怕的还有模糊和过度简化。纯粹主义者经常嘲笑这种大众化的努力方向, 他们喜欢将“原始结果”以一种只有专家才能理解的方式表达出来。我们的理念与他们不同: 将不平凡的、有价值的思想灌输到初学者的头脑中是一件很困难的事情, 对于提高我们的专业水准是很有帮助的, 也能最大程度为社会做出贡献。

多年以来, Brian Kernighan 参与了很多有影响的程序设计和出版项目。两个典型例子是 AWK——一种早期的脚本语言, 用作者名字的首字母命名(Aho、Weinberger 和 Kernighan), 以及 AMPL(A Mathematical Programming Language, 数学编程语言)。

目前, Brian Kernighan 是普林斯顿大学的教授; 他是一位优秀的教师, 擅长于将复杂问题讲得清晰易懂。他为贝尔实验室计算机科学研究中心工作超过了 30 年。贝尔实验室后来更名为 AT&T 贝尔实验室, 然后又被拆分为 AT&T 实验室和朗讯贝尔实验室。他在多伦多大学获得了物理学学士学位, 在普林斯顿大学获得了电子工程博士学位。

C 语言的家族树结构如下图:



C 源于三种语言: 英国的一直未完成的项目 CPL; Martin Richards 离开剑桥大学访问麻省理工学院时设计的 BCPL (Basic CPL) 语言; 以及由 Ken Thompson 设计的称为 B 的解释型语言。后来, C 制订了 ANSI 和 ISO 标准, 并有很多特性受到了 C++ 的影响 (如函数参数检查和 const)。

CPL 是剑桥大学和伦敦的帝国理工学院的合作项目。这个项目最初是在剑桥大学进行的, 因此“C”的正式含义是“剑桥”(Cambridge)。当帝国理工学院参与进来后, “C”的正式含义就变为“联合”(Combined)。实际上 (我们常常听说的) 它一直就表示“Christopher”, 即 CPL 的主要设计者 Christopher Strachey。

参考文献

- Brian Kernighan's home page: <http://cm.bell-labs.com/cm/cs/who/bwk>.
 Dennis Ritchie's home page: <http://cm.bell-labs.com/cm/cs/who/dmr>.
 ISO/IEC 9899:1999. *Programming Language - C*. (The C standard.)
 Kernighan, Brian, and Dennis Ritchie. *The C Programming Language*. Prentice Hall, 1978. Second Edition, 1989. ISBN 0131103628.
 A list of members of the Bell Labs' Computer Science Research Center: <http://cm.bell-labs.com/cm/cs/alumni.html>.
 Ritchards, Martin. *BCPL - The Language and Its Compiler*. Cambridge University Press, 1980. ISBN 052121-9655.
 Ritchie, Dennis. "The Development of the C Programming Language. Proceedings of the ACM History of Programming Languages Conference (HOPL-2). *ACM SIGPLAN Notices*, Vol. 28 No. 3, 1993.
 Salus, Peter. *A Quarter Century of UNIX*. Addison-Wesley, 1994. ISBN 0201547775.

22.2.6 C++

C++ 是一种偏向于系统编程的通用程序设计语言, 它的特点是:

- 可以看做是更好的 C
- 支持数据抽象
- 支持面向对象程序设计
- 支持泛型程序设计

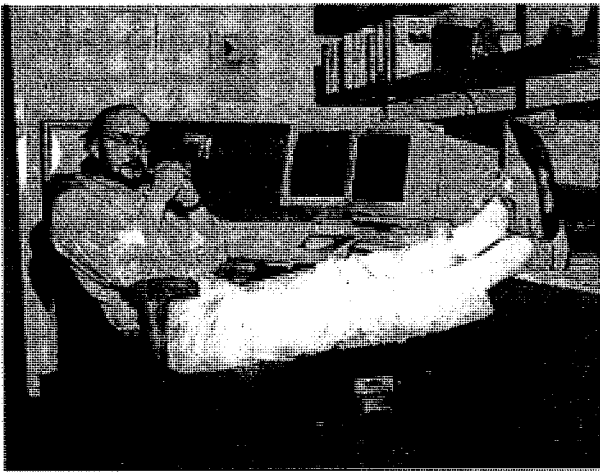
C++ 最初是由 Bjarne Stroustrup 在新泽西州茉莉山的贝尔电话实验室计算科学研究中心设计并实现的。他的办公室与 Dennis Ritchie、Brian Kernighan、Ken Thompson、Doug McIlroy 以及其他 UNIX 巨人们相邻。

Bjarne Stroustrup 在家乡的丹麦奥胡斯大学数学专业获得了计算机科学硕士学位。然后, 他来到剑桥大学, 在 David Wheeler 指导下获得了计算机科学博士学位。C++ 的主要贡献包括:

- 使抽象技术对于主流项目来说, 其应用代价可以承受, 易于管理。
- 将面向对象和泛型程序设计技术用于对性能有较高要求的应用领域的先驱。

在 C++ 出现之前, 这些技术 (通常一起混杂在“面向对象程序设计”的标签之下) 并不为工业界所熟知。与 Fortran 之前的科学计算程序设计和 C 之前的系统程序设计所处的情况类似, 人们“公

认”这些技术对于实际应用来说代价太高，对于“普通程序员”来说太难以掌握。



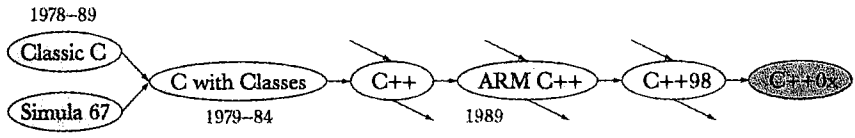
C++ 的研究工作始于 1979 年，在 1985 年发布了第一个商用版本。在最初的设计和实现之后，Bjarne Stroustrup 与贝尔实验室和其他地方的朋友们进一步对其进行完善，直到 1990 年 C++ 标准化进程正式开始。从那时起，C++ 的定义由 ANSI(美国国家标准化组织)负责制订，从 1991 年起改由 ISO(国际标准化组织)负责。Bjarne Stroustrup 在这一进程中承担了主要工作，他是负责语言新特性的关键小组的主席。第一个国际标准(C++ 98)在 1998 年批准通过，第二个版本(C++ 0x)正在制订中。

经过最初十年的成长，C++ 最重要的发展就是 STL——容器和算法的标准库。它主要是 Alexander Stepanov 几十年努力的成果，其目标是生成更通用、更有效的软件，它从数学之美、数学之实用中受到了很多启发。

Alex Stepanov 是 STL 的发明者和泛型程序设计的先驱。他毕业于莫斯科大学，研究工作包括机器人、算法及其他领域，他在这些工作中使用过多种语言(包括 Ada、Scheme 和 C++)。从 1979 年开始，他在美国学术和工业界工作，曾为通用电气实验室、AT&T 贝尔实验室、惠普、Silicon Graphics 和 Adobe 等工作。



C++ 语言家族树如下图所示：



Bjarne Stroustrup 最初的设想是“支持类的 C”——综合 C 和 Simula 的思想。但随着其后继者 C++ 的实现，这一设想就消失了。

人们讨论程序设计语言时，常常集中在优雅的设计和高级特性上。但是，以这两方面评判，C 和 C++ 并不是计算领域中历史上最成功的两种语言。它们的强大在于灵活性、性能和稳定性。重要的软件系统的生命期会超过几十年，它们通常会耗尽硬件资源，并且还常常遇到完全无法预料的修改需求。C 和 C++ 已经证明了它们能在这种环境中茁壮成长。我们喜欢引用 Dennis Ritchie 的名言：

“有些语言是为了证明一个观点而设计，而其他一些语言则是为了解决问题。”这里的“其他”主要是指 C。Bjarne Stroustrup 喜欢说：“其实我知道如何设计一种比 C++ 更漂亮的语言。”C++ 与 C 一样，其目标不是抽象的美（尽管我们很赞成在可能的情况下追求美）而是实用。

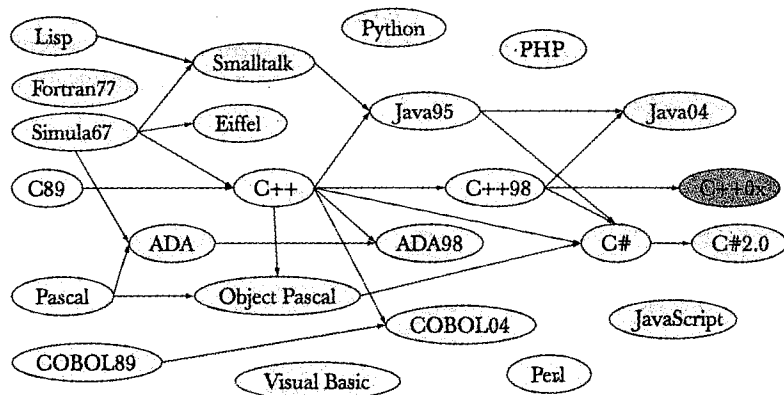
我经常后悔在本书中没有使用 C++0x 的特性。它本可以简化很多例子和解释。不过，书中已经介绍了一些 C++0x 标准库的特性：unordered_map(参见 21.6.4 节)、array(参见 20.9 节)和 regex(参见 23.5 ~ 23.9 节)。C++0x 还有其他一些特点：更好的模板检查、更简单更通用的初始化以及某些地方更健壮的表达法。请参考我在 HOPL-III 上的论文。

参考文献

Alexander Stepanov's publications: www.stepanovpapers.com.
 Bjarne Stroustrup's home page: www.research.att.com/~bs.
 ISO/IEC 14882:2003. *Programming Languages - C++*. (The C++ standard.)
 Stroustrup, Bjarne. "A History of C++: 1979-1991. Proceedings of the ACM History of Programming Languages Conference (HOPL-2). *ACM SIGPLAN Notices*, Vol. 28 No. 3, 1993.
 Stroustrup, Bjarne. *The Design and Evolution of C++*. Addison-Wesley, 1994. ISBN 0201543303.
 Stroustrup, Bjarne. *The C++ Programming Language (Special Edition)*. Addison-Wesley, 2000. ISBN 0201700735.
 Stroustrup, Bjarne. "C and C++: Siblings"; "C and C++: A Case for Compatibility"; and "C and C++: Case Studies in Compatibility." *The C/C++ Users Journal*. July, Aug., and Sept. 2002.
 Stroustrup, Bjarne. "Evolving a Language in and for the Real World: C++ 199 - 2006. Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages (HOPL-III). San Diego, CA, 2007. <http://portal.acm.org/toc.cfm?id=1238844>.

22.2.7 今天的程序设计语言

目前还在使用的程序设计语言有哪些，用于什么领域？这确实是一个难以回答的问题。当前语言的家族树——即便是以最简化的形式呈现，也很拥挤、杂乱，如下所示：



实际上，我们在互联网上(或其他地方)找到的大多数统计数据都比谣言强不了多少，因为它们所选取的都是一些与语言的使用关联度很差的指标。例如，不少网站张贴的内容包括程序语言的名词、编译器发货量、学术论文量和书籍销售量等。但所有这些指标都更倾向于新语言而不是已成熟的语言。我们再换个问题，什么人可以称做程序员呢？每天都使用程序设计语言的人？只是在学习时写些小程序的学生算是程序员吗？讲授程序设计的教授呢？几乎每年都只写一个程序的物理学家呢？如果一个专业程序员每周都会使用几种不同的语言，那么他应该被统计多次还是一次呢？我们可以看到，对这些问题的每个答案都会导致不同的统计方式和结果。

但是，我们认为有必要给你一个具体意见：2008 年全世界大约有 1000 万专业程序员。我们

是从 IDC(一个数据收集公司)的数据、与出版商和编译器提供商进行的讨论以及各种互联网资源得出这个结论的。这个数量可能不准确,但我们确定,无论“程序员”定义如何,只要大致合理,实际值肯定在 100 万到 1 亿之间。程序员们使用哪种语言呢? Ada、C、C++、C#、COBOL、Fortran、Java、PERL、PHP 以及 Visual Basic 可能(仅仅是可能)占据了超过 90% 的份额。

除了本章提到的语言之外,我们还可以列出几十甚至上百种语言。但除了对感兴趣的和重要的语言公平些以外,这没有任何意义。如果需要的话,你可以自行查找相关信息。一个专业人员通常都懂几种语言,并且有能力在必要时学习新的语言。世界上并不存在适用于所有人和所有应用的“真正的语言”。实际上,我们所能想到的所有重要系统,都使用了不止一种语言来实现。

22.2.8 参考资源

上面提到的每种语言都有一个参考资料列表。下面是一些涵盖几种语言的参考资料:

更多的语言设计者的网页/图片

www.angelfire.com/tx4/cus/people/.

几个语言例子

<http://dmoz.org/Computers/Programming/Languages/>

教科书

Scott, Michael L. *Programming Language Pragmatics*. Morgan Kaufmann, 2000. ISBN 1558604421.

Sebesta, Robert W. *Concepts of Programming Languages*. Addison-Wesley, 2003. ISBN 0321193628.

历史书籍

Bergin, T. J., and R. G. Gibson, eds. *History of Programming Languages - II*. AddisonWesley, 1996. ISBN 0201895021.

Hailpern, Brent, and Barbara G. Ryder, eds. *Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages (HOPL-III)*. San Diego, CA, 2007. <http://portal.acm.org/toc.cfm?id=1238844>.

Lohr, Steve. *Go To: The Story of the Math Majors, Bridge Players, Engineers, Chess Wizards, Maverick Scientists and Iconoclasts - The Programmers Who Created the Software Revolution*. Basic Books, 2002. ISBN 9780465042265.

Sammet, Jean. *Programming Languages: History and Fundamentals*. Prentice-Hall, 1969. ISBN 0137299885.

Wexelblat, Richard L., ed. *History of Programming Languages*. Academic Press, 1981. ISBN 0127450408.



思考题

1. 历史有什么用处?
2. 程序设计语言有什么用处? 请给出几个例子。
3. 请给出几种可以认为是客观优点的程序设计语言的基本特性。
4. 抽象的含义是什么? 更高层的抽象呢?
5. 我们对程序设计的 4 个高层的理念是什么?
6. 列出高层程序设计的潜在优点。
7. 什么是重用? 它可以带来什么好处?
8. 什么是过程式程序设计? 给出一个具体的例子。
9. 什么是数据抽象? 给出一个具体的例子。
10. 什么是面向对象程序设计? 给出一个具体的例子。
11. 什么是泛型程序设计? 给出一个具体的例子。
12. 什么是多范型程序设计? 给出一个具体的例子。
13. 第一个运行在储存程序式计算机上的程序出现在什么时候?
14. David Wheeler 以什么工作知名?
15. John Backus 设计的第一种语言的主要贡献是什么?
16. Grace Murray Hopper 设计的第一种语言是什么?

17. John McCarthy 的主要研究领域是什么?
18. Peter Naur 对 Algol60 有哪些贡献?
19. Edsger Dijkstra 以什么工作知名?
20. Niklaus Wirth 设计并实现的是哪种语言?
21. Anders Heijlsberg 设计的是哪种语言?
22. Jean Ichbiah 在 Ada 项目中扮演什么角色?
23. Simula 开创了什么程序设计风格?
24. Kristen Nygaard 曾在哪里(奥斯陆之外)教书?
25. Ole-Johan Dahl 以什么工作知名?
26. Ken Thompson 是哪个操作系统的主要设计者?
27. Doug McIlroy 以什么工作知名?
28. Brian Kernighan 最著名的著作是什么?
29. Dennis Ritchie 曾在哪里工作?
30. Bjarne Stroustrup 以什么工作知名?
31. Alex Stepanov 使用哪种语言设计了 STL?
32. 列出 10 种 22.2 节中没有介绍的语言。
33. Scheme 是哪种语言的方言?
34. C++ 最主要的两个起源是什么?
35. C++ 语言中的 C 表示的是什么?
36. Fortran 是一个缩写吗? 如果是, 全称是什么?
37. COBOL 是一个缩写吗? 如果是, 全称是什么?
38. Lisp 是一个缩写吗? 如果是, 全称是什么?
39. Pascal 是一个缩写吗? 如果是, 全称是什么?
40. Ada 是一个缩写吗? 如果是, 全称是什么?
41. 哪种编程语言最好?

术语

在本章中, “术语”是真实的语言、人和组织。

• 语言

Ada	Algol	BCPL	C	C++	COBOL
Fortran	Lisp	Pascal	Scheme	Simula	

• 人

Charles Babbage	John Backus	Ole-Johan Dahl	Edsger Dijkstra	Anders Heijlsberg
Grace Murray Hopper	Jean Ichbiah	Brian Kernighan	John McCarthy	
Doug McIlroy	Peter Naur	Kristen Nygaard	Dennis Ritchie	
Alex Stepanov	Bjarne Stroustrup	Ken Thompson	David Wheeler	
Niklaus Wirth				

• 组织

贝尔实验室	Borland 公司	剑桥大学(英国)
ETH(苏黎世联邦理工学院)	IBM 公司	麻省理工学院
挪威计算机中心	普林斯顿大学	斯坦福大学
哥本哈根理工大学	美国国防部	美国海军

习题

1. 请给出程序设计的定义。
2. 请给出程序设计语言的定义。

3. 浏览本书，注意章节中的插图。哪些插图是计算机科学家？编写一段文字，总结每位科学家的贡献。
4. 浏览本书，注意章节中的插图。哪些插图不是计算机科学家？指出每幅插图出自哪个国家、哪个领域。
5. 用本章提到的每种语言各编写一个“Hello, World”程序。
6. 对于本章提到的每种语言，找到一本流行的教科书，查找其中使用的第一个完整程序。用所有其他语言编写这个程序。警告：这个练习的规模很容易达到 100 个程序。
7. 我们明显“遗漏”了很多重要的语言。特别是，我们基本没有提及 C++ 之后程序设计语言的发展。列出你认为应该介绍的 5 种现代语言，然后沿着本章语言部分的路线，用一页半的篇幅介绍其中 3 种。
8. C++ 有什么用处？为什么？撰写一份 10~20 页的报告。
9. C 有什么用处？为什么？撰写一份 10~20 页的报告。
10. 针对一种语言（不包括 C 和 C++），撰写一份 10~20 页的报告，描述这种语言的起源、目标和功能。给出足够的具体例子。介绍谁在使用这种语言以及用它做什么？
11. 谁是剑桥大学现任的卢卡斯教授？
12. 本章提到的语言设计者中，谁拥有数学学位？谁没有？
13. 本章提到的语言设计者中，谁拥有博士学位？在哪个领域？谁没有博士学位？
14. 本章提到的语言设计者中，谁获得过图灵奖？图灵奖是什么？查找这些人是因何贡献而获奖。
15. 编写一个程序，输入一个由 (name, year) 值对构成的文件，例如 (Algol, 1960) 和 (C, 1974)，并在时间轴上画出这些名字。
16. 修改上一个练习的程序，改为读取由 (name, year, (ancestors)) 三元组组成的文件，例如 (Fortran, 1956, ())、(Algol, 1960, (Fortran)) 和 (C++, 1985, (C, Simula))，在时间轴中画出这些语言，并在每个祖先和后代之间画一个箭头。用这个程序画出 22.2.2 节和 22.2.7 节中的图表的改进版本。

附言

很明显，我们只是泛泛地介绍了程序设计语言的历史和如何开发更好的软件的理念。我们认为历史和理念非常重要，以史为鉴能使你体会到什么是错误的。我们希望本章的内容已经传达出了一些令我们激动的东西，传达出了我们的观点——好的软件/好的程序设计方法这一问题浩瀚无边，程序设计语言的设计和实现已经充分表明了这一点。请记住，程序设计（开发高质量的软件）才是最基础、最重要的；程序设计语言只是工具。

第23章 文本处理

“所谓显然的事情通常并非真的那么显然……
使用‘显然’这个词往往意味着缺乏逻辑论证。”

——Errol Morris

本章主要介绍如何从文本中提取信息。我们将大量知识以单词的形式保存在文档中，例如书籍、电子邮件或者“打印”的表格，以便将来能从中提取出某种更易于计算的信息格式。在本章中，我们首先回顾标准库中最常用的文本处理功能：string、iostream 以及 map。然后，我们将介绍正则表达式(regex)，它可以用来描述文本中的模式。最后，我们将展示如何使用正则表达式从文本中寻找和提取特定的数据元素，如邮政编码，以及如何验证文本文件的格式。

23.1 文本

从本质上来说，我们无时无刻不在处理文本。我们阅读的书籍中全都是文本，我们在电脑屏幕上看到的很多内容都是文本，我们处理的源代码也是文本。我们使用的(所有)通信信道充斥着文本。两个人之间的所有交流内容也都可以表示为文本。但我们不要走极端，图像和声音通常还是表示为二进制格式(即比特包)更好些。不过，对于几乎所有其他信息，都适合使用计算机程序进行文本分析和转换。

从第3章开始，我们就已经看到了 iostream 和 string 的使用。因此，在本章中，我们只是简单回顾一下这些标准库中的语言特性。标准库中的“映射”(参见 23.4 节)是一种非常有用的文本处理工具，我们将以电子邮件分析问题为例展示映射的使用。在回顾了这些标准库特性后，我们将重点介绍如何使用正则表达式搜索文本中的模式(参见 23.3 ~ 23.10 节)。

23.2 字符串

一个字符串(string)包含一个字符序列，并提供了一些有用的操作，如向字符串中添加一个字符、获得字符串的长度以及字符串连接等。实际上，标准库中的字符串提供了很多操作，但其中大部分只用于相当复杂的低层方式的文本处理。在本章中，我们只简单回顾少数最常用的操作。如果需要的话，你可以在参考手册或者专业级的教材中查阅这些操作(以及全部字符串操作)的细节。这些操作的定义可以在 <string> 中找到(注意不是 <string.h>)：

挑选出的一些字符串操作

s1 = s2	将 s2 的内容赋予 s1, s2 可以是字符串对象或者 C 风格字符串
s += x	将 x 添加到 s 的末尾, x 可以是字符、字符串或者 C 风格字符串
s[i]	数组下标(s 的第 i 个字符)
s1 + s2	连接运算, 结果字符串的开始部分拷贝自 s1, 后接 s2 的拷贝
s1 == s2	比较字符串的值, s1 或 s2 可以是 C 风格字符串, 但不允许两者皆是。!= 也类似
s1 < s2	按字典序比较两个字符串的先后, 两者之一可以是 C 风格字符串, 但不允许两者皆是。 <=、> 和 >= 也类似
s.size()	s 中字符的数目

挑选出的一些字符串操作

<code>s.length()</code>	<code>s</code> 中字符的数目
<code>s.c_str()</code>	返回 <code>s</code> 中字符构成的 C 风格字符串
<code>s.begin()</code>	指向第一个字符的迭代器
<code>s.end()</code>	指向 <code>s</code> 末尾之后一个位置的迭代器
<code>s.insert(pos, x)</code>	将 <code>x</code> 插入到 <code>s[pos]</code> 之前的位置。 <code>x</code> 可以是字符、字符串或者 C 风格字符串。必要时会扩展 <code>s</code> 的存储空间来容纳 <code>x</code>
<code>s.append(pos, x)</code>	将 <code>x</code> 插入到 <code>s[pos]</code> 之后的位置。 <code>x</code> 可以是字符、字符串或者 C 风格字符串。必要时会扩展 <code>s</code> 的存储空间来容纳 <code>x</code>
<code>s.erase(pos)</code>	删除 <code>s[pos]</code> 处的字符。 <code>s</code> 的长度减小 1
<code>pos = s.find(x)</code>	在 <code>s</code> 中查找 <code>x</code> 。 <code>x</code> 可以是字符、字符串或者 C 风格字符串。若找到, <code>pos</code> 的值为找到的字符串的第一个字符的下标, 否则为 <code>npos</code> (<code>s</code> 末尾之后的位置)
<code>in >> s</code>	从流 <code>in</code> 中读取一个词存入 <code>s</code> 中, 词之间以空白符间隔
<code>getline(in, s)</code>	从流 <code>in</code> 中读取一行存入 <code>s</code>
<code>out << s</code>	将 <code>s</code> 的内容输出到 <code>out</code>

我们已经在第 10 章和第 11 章中介绍了 I/O 操作, 在 23.3 节中将进行小结。注意输入操作在必要时会扩展字符串的存储空间, 因此可能发生溢出。

`insert()` 和 `append()` 操作会移动已有字符, 为新字符留出空间。`erase()` 操作则“向前”移动字符, 避免删除字符后在字符串中留下空洞。

标准库字符串实际上是一个称为 `basic_string` 的模板, 它支持多种字符集, 如 Unicode(在“普通字符”之外还提供了几千个特殊字符, 如 £、Ω、∞、δ、☺ 以及 ♪)。例如, 如果你需要声明一个保存 Unicode 字符的类型, 其名字就叫做 Unicode, 可以使用如下代码:

```
basic_string<Unicode> a_unicode_string;
```

我们之前所使用的标准字符串类 `string`, 实际上就是保存普通字符的 `basic_string`:

```
typedef basic_string<char> string; // string means basic_string<char>
```

本章不会介绍 Unicode 字符或 Unicode 字符串, 如果你需要使用这些功能的话, 可以查阅相关资料, 你会发现其使用方法(从语言的角度, 以及从 `string`、`iostream` 和正则表达式的角度)与普通字符和普通字符串是一致的。如果你需要使用 Unicode 字符, 最好求助于有经验的人。编写这类程序, 除了程序设计语言方面的考虑外, 还要遵循系统方面的惯例。

在文本处理语境中, 将几乎所有内容都表示为字符串是很重要的。例如, 在本页中, 12.333 表示为 6 个字符的字符串(前后都有空白符)。如果我们要读取这个数, 就必须将这 6 个字符转换为一个浮点型数, 然后才能进行算术运算。这就要求提供两个方向的转换功能: 值转换为字符串和字符串转换为值。在 11.4 节中, 我们已经看到了如何利用 `stringstream` 将一个整数转换为一个字符串。这种技术可以推广到任何具有 `<<` 操作符的类型:

```
template<class T> string to_string(const T& t)
{
    ostringstream os;
    os << t;
    return os.str();
}
```

其使用方式如下例:

```
string s1 = to_string(12.333);
string s2 = to_string(1+5*6-99/7);
```

这两行代码执行后，s1 的值变为“12.333”，s2 的值变为“17”。实际上，to_string() 不仅可以用于数值类型，还可用于其他任何具有 << 操作符的类型 T。与之相反的转换，也就是从 string 到数值的转换，也同样简单、有用：

```
struct bad_from_string : std::bad_cast
    // class for reporting string cast errors
{
    const char* what() const    // override bad_cast's what()
    {
        return "bad cast from string";
    }
};

template<class T> T from_string(const string& s)
{
    istream is(s);
    T t;
    if (!is >> t) throw bad_from_string();
    return t;
}
```

使用示例如下：

```
double d = from_string<double>("12.333");

void do_something(const string& s)
try
{
    int i = from_string<int>(s);
    // ...
}
catch (bad_from_string e) {
    error("bad input string",s);
}
```

与 to_string() 相比，from_string() 要稍微复杂一些：同一个字符串所表示的值，可以解释为很多类型。这也意味着，我们所看到的字符串内容，未必表示我们所期待的类型的值。例如：

```
int d = from_string<int>("Mary had a little lamb");    // oops!
```

这样就可能引起错误，我们用异常 bad_from_string 来表示这类错误。在 23.9 节中，我们将说明为什么 from_string() (或等价的函数) 对文本处理那么重要，其原因就在于我们需要从文本域中提取数值。在 16.4.3 节中，我们已经看到一个类似的函数 get_int() 在 GUI 代码中的应用。

注意，to_string() 和 from_string 这两个函数是非常相似的。实际上，他们大致互为逆操作。这样，对于所有“适当的类型 T”，我们有如下结论(忽略空白符、舍入等细节)：

```
s==to_string(from_string<T>(s))    // for all s
```

以及

```
t==from_string<T>(to_string(t))    // for all t
```

“适当”的意思是指 T 应该具有默认构造函数、>> 操作符以及一个匹配的 << 操作符。

注意，to_string() 和 from_string() 的实现中都使用了一个 stringstream 来完成所有困难的工作。实际上，对于任何具有匹配的 << 和 >> 操作符的类型，我们都可以利用这种技术手段来实现类型之间的转换操作：

```
struct bad_lexical_cast : std::bad_cast
{
    const char* what() const { return "bad cast"; }
};

template<typename Target, typename Source>
Target lexical_cast(Source arg)
{
    std::stringstream interpreter;
    Target result;

    if (!(interpreter << arg)           // read arg into stream
        || !(interpreter >> result)    // read result from stream
        || !(interpreter >> std::ws).eof()) // stuff left in stream?
        throw bad_lexical_cast();

    return result;
}
```

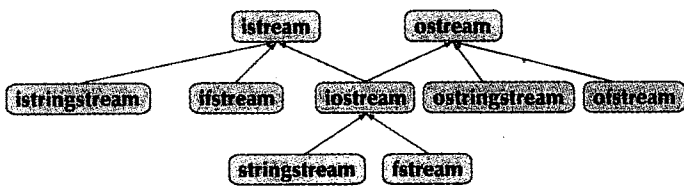
!(interpreter >> std::ws).eof() 看起来有些怪，但实际上是很巧妙的，它将提取数据后可能遗留在 stringstream 中的空白符读取出来。空白符是允许出现的，但在它之后不允许再有任何字符，我们可以通过查看是否到达“文件尾”来检测这一情况。这样，当我们试图用 lexical_cast 从一个 string 对象中读取一个 int 值时，“123”和“123 ”都会成功读取数值，而“123 5”会失败，因为空格之后还有一个字符 5。

23.3 I/O 流

如上节所示，我们利用 I/O 流在字符串和其他类型间建立起联系。I/O 流库不仅仅可以实现输入和输出，它还可以实现字符串格式和内存类型之间的转换。标准库 I/O 流提供了对字符串进行读、写和格式化的功能。我们已经在第 10 章和第 11 章介绍了 iostream 库，因此现在只是简单总结一下：

I/O 流	
in >> x	根据 x 的类型从 in 读取数据存入 x
out << x	根据 x 的类型将其内容写入 out
in.get(c)	从 in 读取一个字符存入 c
getline(in, s)	从 in 读取一行内容存入字符串 s

标准流的类层次如下图所示(参见 14.3 节)：



这些类一起构成了标准流库，使我们可以对文件和字符串进行 I/O 操作(还可对其他任何可以看做文件或字符串的对象进行 I/O 操作，如键盘、屏幕等，参见第 10 章)。而且，如第 10 章和第 11 章所述，iostream 还提供了精心设计的格式化机制。上图中，箭头表示继承关系(参见 14.3 节)，因而，一个 stringstream 对象可以作为一个 iostream 或者 istream 或 ostream 来使用。

与字符串类似, `iostream` 可以使用 Unicode 这样的大字符集, 用法与普通字符集一致。我们再次提醒, 如果你需要使用 Unicode I/O, 最好求助于有经验的人。为了正确使用 Unicode, 你的程序除了要考虑程序设计语言方面的问题外, 还要遵循系统方面的惯例。

23.4 映射

关联数组 (associative array)——包括映射、散列表 (hash table) 等数据结构, 是很多文本处理的关键 (key 为双关语, 有“关键”之意, 在关联数组类型中, 它又表示与数据关联的“关键字”)。原因很简单: 当我们处理文本时, 主要工作就是收集信息, 而信息通常与文本字符串 (如名字、地址、邮政编码、社会保险号、职位等) 相关联。即使某些文本字符串可以转换为数值, 但通常更方便也更简单的方式还是将它们按文本来处理, 并使用此文本作为信息的标识。21.6 节中的单词计数的例子是一个很好的简单示例。如果你还不太习惯使用映射, 请重新阅读 21.6 节, 然后再继续学习本节的内容。

下面我们以电子邮件问题为例, 来讨论映射在文本处理中的应用。我们常常需要搜索和分析电子邮件消息和邮件日志, 这一般要借助一些专门程序 (如 Thunderbird 或 Outlook) 来完成。利用这些程序, 我们不必查看海量的完整邮件内容, 就能找到所需要的信息。然而, 通常我们要查找的信息, 如发件人、收件人、发送地址以及其他更多的内容, 都是以邮件头中文本的形式呈现给邮件程序的。这几乎就已经是完整的邮件数据了, 所以邮件程序必须能够从这些海量数据中搜索到我们要找的内容。目前, 分析邮件头的工具有上千种, 大多数都利用正则表达式 (将在 23.5 ~ 23.9 节中进行详细介绍) 进行信息提取, 并生成与相关邮件相联系的某种形式的关联数组。例如, 我们常常需要查找某个人发送来的所有邮件, 或者同一主题的所有邮件, 或者内容与特定主题相关的所有邮件。

在这里, 我们使用一个非常简单的邮件文件来展示一些从文本文件中提取数据的技术。这个邮件文件的头是来自于 www.faqs.org/rfcs/rfc2822.html 的实际的 RFC2822 头, 如下所示:

```
xxx
xxx
-----
From: John Doe <jdoe@machine.example>
To: Mary Smith <mary@example.net>
Subject: Saying Hello
Date: Fri, 21 Nov 1997 09:55:06 -0600
Message-ID: <1234@local.machine.example>

This is a message just to say hello.
So, "Hello".
-----
From: Joe Q. Public <john.q.public@example.com>
To: Mary Smith <@machine.tld:mary@example.net>, jdoe@test.example
Date: Tue, 1 Jul 2003 10:52:37 +0200
Message-ID: <5678.21-Nov-1997@example.com>

Hi everyone.
-----
To: "Mary Smith: Personal Account" <smith@home.example>
From: John Doe <jdoe@machine.example>
Subject: Re: Saying Hello
Date: Fri, 21 Nov 1997 11:00:00 -0600
Message-ID: <abcd.1234@local.machine.tld>
In-Reply-To: <3456@example.net>
```

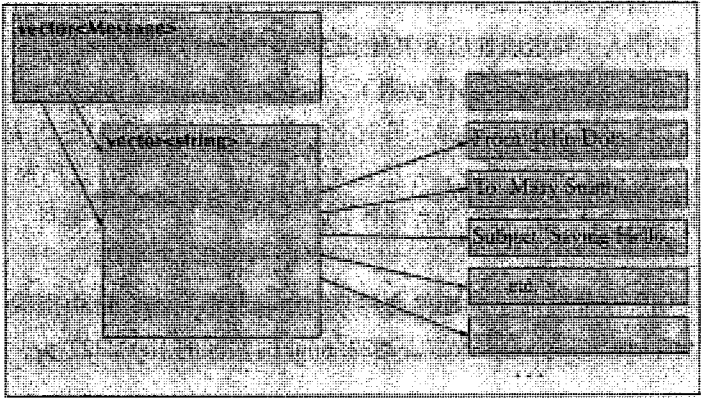
References: <1234@local.machine.example> <3456@example.net>

This is a reply to your reply.

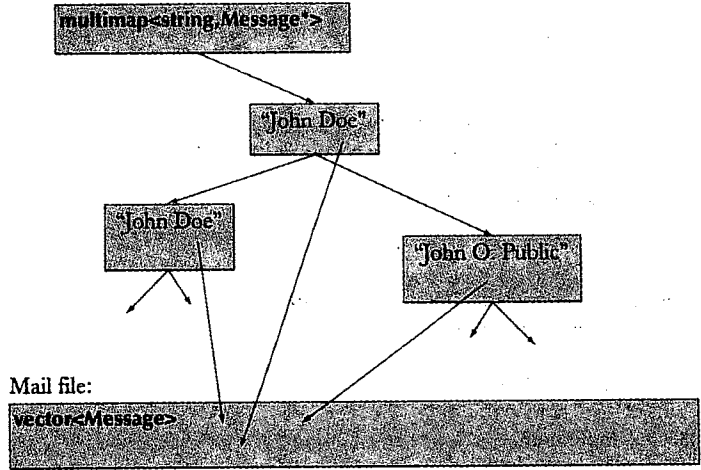
简单起见，我们已经丢弃了邮件文件中的大部分内容。我们还在每个邮件之后添加了一行“-----”(4个连字符)，来标识邮件的结束，这进一步简化了邮件的分析。我们将会写一个简短的“玩具程序”，它查找所有“John Doe”发送的邮件，并输出这些邮件的主题。这个程序虽然简短，但其中的技术可以用来完成很多更复杂、更有趣的任务。

首先，我们要确定是要随机访问数据，还是通过一个输入流以流的方式分析数据。我们选择第一种方式，因为在一个实际程序中，我们可能只关心某几个发件人，或者来自于一个发件人的某些信息。而且，相对于流式访问，随机访问确实更困难些，因此我们可以学习更多技术。特别地，我们将再次使用迭代器。

我们的基本思路是读取一个完整的邮件文件，将其内容保存到一个称为 Mail_file 的数据结构中。这个数据结构用一个向量 `vector<string>` 保存邮件文件的所有文本行，并通过另一个向量 `vector<Message>` 指明每个邮件的起止位置，如下所示：



为了实现这一目的，我们将加入迭代器和 `begin()`、`end()` 函数，以便能有一种一致的方法遍历所有行和所有邮件。通过这个“样板代码”，我们可以方便地访问邮件。完成它之后，我们就可以编写我们的“玩具程序”了，这个程序将每个发件人的所有邮件收集在一起，以方便访问，如下所示：



最终，我们输出所有发自“John Doe”的邮件的主题内容，以此来展示如何使用这套工具程序来处理邮件。

编写这个程序，需要使用很多基础的标准库功能，如下所示：

```
#include<string>
#include<vector>
#include<map>
#include<fstream>
#include<iostream>
using namespace std;
```

我们将 Message 定义为 `vector<string>`（保存文本行的向量）的一对迭代器如下所示：

```
typedef vector<string>::const_iterator Line_iter;
class Message { // a Message points to the first and the last lines of a message
    Line_iter first;
    Line_iter last;
public:
    Message(Line_iter p1, Line_iter p2) : first(p1), last(p2) { }
    Line_iter begin() const { return first; }
    Line_iter end() const { return last; }
    // ...
};
```

我们定义一个 Mail_file 类，保存文本行和邮件，如下所示：

```
typedef vector<Message>::const_iterator Mess_iter;

struct Mail_file { // a Mail_file holds all the lines from a file
    // and simplifies access to messages
    string name; // file name
    vector<string> lines; // the lines in order
    vector<Message> m; // Messages in order

    Mail_file(const string& n); // read file n into lines

    Mess_iter begin() const { return m.begin(); }
    Mess_iter end() const { return m.end(); }
};
```

注意，我们是如何将迭代器加入数据结构中，以便能更容易地、更有条理地访问数据结构的内容。我们这里并没有真正使用标准库算法，但由于使用了迭代器，程序很容易改为使用标准库算法的版本。

为了在邮件中查找和提取信息，我们需要下面两个辅助函数：

```
// find the name of the sender in a Message;
// return true if found
// if found, place the sender's name in s:
bool find_from_addr(const Message* m, string& s);

// return the subject of the Message, if any, otherwise "":
string find_subject(const Message* m);
```

最后，我们就可以编写从文件提取信息的代码了，如下所示：

```
int main()
{
    Mail_file mfile("my-mail-file.txt"); // initialize mfile from a file

    // first gather messages from each sender together in a multimap:

    multimap<string, const Message*> sender;
```

```

    for (Mess_iter p = mfile.begin(); p!=mfile.end(); ++p) {
        const Message& m = *p;
        string s;
        if (find_from_addr(&m,s))
            sender.insert(make_pair(s,&m));
    }

    // now iterate through the multimap
    // and extract the subjects of John Doe's messages:
    typedef multimap<string, const Message*>::const_iterator MCI;
    pair<MCI,MCI> pp = sender.equal_range("John Doe");
    for(MCI p = pp.first; p!=pp.second; ++p)
        cout << find_subject(p->second) << '\n';
}

```

我们来仔细分析一下程序中是如何使用映射的。我们使用了一个 multimap (参见 20.10 节和附录 B.4)，因为我们希望将来自于同一个地址的很多邮件收集到一个地方存储。标准库的 multimap 就可以完成这一任务 (使得访问具有相同关键字的数据更为容易)。显然，我们的工作分为两部分 (一般情况下并不是这样)：

- 创建映射。
- 使用映射。

我们遍历所有邮件，用 insert() 将它们插入到 multimap 中，从而创建了 multimap 的内容，如下所示：

```

    for (Mess_iter p = mfile.begin(); p!=mfile.end(); ++p) {
        const Message& m = *p;
        string s;
        if (find_from_addr(&m,s))
            sender.insert(make_pair(s,&m));
    }

```

插入到映射中的内容都是我们用 make_pair() 构造的 (关键字, 值) 对。我们使用“自制的” find_from_addr() 函数查找发件人的名字，这里使用空字符串表示查找失败。

我们在程序中使用了引用类型的变量 m，令其引用 p 所指向的对象，又将其地址传递给函数 find_from_addr()，为什么不直接将 p 传递给 find_from_addr(p,s) 呢？这是因为，虽然我们知道 Mess_iter 指向一个 Message 对象，但具体实现中它不一定是一个指针，所以间接利用 m 保证传递给 find_from_addr() 的是一个指针。

在程序中，我们首先将所有 Message 对象存于一个 vector 中，然后又通过这个 vector 创建了一个 multimap，为什么不直接将 Message 对象存入 multimap 中呢？原因很简单，这也是一个数据处理的基本原则：

- 首先，我们创建一个通用的数据结构，可利用它来完成很多工作。
- 然后，我们将它转换为用于特定目的的专用结构。

通过这种方法，我们可以创建一个可重用组件的集合。反之，如果我们直接创建一个 multimap 的话，在希望完成其他任务时，可能就需要对其进行重新定义。特别是，我们的 multimap 对象 (称为 senders) 是按照邮件的地址域进行排序的。而对于其他很多应用来说，这种顺序可能毫无意义，它们可能关心返回地址、收件人、抄送地址、主题、时间戳等。

这种逐步 (或者说逐层) 创建应用程序的方法，可以极大地简化设计、实现、文档和维护工作。关键点在于每个部分都只做一件事，而且是以一种简单直接的方法去做。与之相对的方法，是将所有事情都放在一起做，这种方法需要极高的聪明才智。显然，我们这个“从一个电子邮件

头中提取信息”的小程序，只是逐层构造方法的一个很简单的应用。逐层构造方法的保持独立组件相分离、模块化以及渐进构造的思想，会随着问题规模的增大体现出更大的价值。

为了提取所需信息，我们简单地使用 `equal_range()` 函数查找所有包含关键字“John Doe”的条目。然后遍历它返回的序列 `[first, second)` 中的所有元素，利用 `find_subject()` 提取主题域，如下所示：

```
typedef multimap<string, const Message*>::const_iterator MCI;
```

```
pair<MCI,MCI> pp = sender.equal_range("John Doe");
```

```
for (MCI p = pp.first; p!=pp.second; ++p)
    cout << find_subject(p->second) << '\n';
```

当我们遍历一个映射的元素时，我们得到一个(关键字, 值)对的序列。对于每个对，第一个元素(本例中是 `string` 类型的关键字)称为 `first`，第二个元素称(本例中是 `Message` 类型的值)为 `second` (参见 21.6 节)。

23.4.1 实现细节

显然，我们需要实现前面程序中所使用的函数。我们可以将这个工作作为练习，以节省版面。但我们决定给出这些实现细节，以使这个例子更加完整。`Mail_file` 的构造函数打开邮件文件，并构造 `lines` 和 `m` 两个向量，如下所示：

```
Mail_file::Mail_file(const string& n)
// open file named "n"
// read the lines from "n" into "lines"
// find the messages in the lines and compose them in m
// for simplicity assume every message is ended by a "-----" line
{
    ifstream in(n.c_str()); // open the file
    if (!in) {
        cerr << "no " << n << '\n';
        exit(1); // terminate the program
    }

    string s;
    while (getline(in,s)) lines.push_back(s); // build the vector of lines

    Line_iter first = lines.begin(); // build the vector of Messages
    for (Line_iter p = lines.begin(); p!=lines.end(); ++p) {
        if (*p == "-----") { // end of message
            m.push_back(Message(first,p));
            first = p+1; // ----- not part of message
        }
    }
}
```

这段程序中的错误处理代码是不完整的。如果我们希望这个程序真正能够使用，还需要进一步完善。

试一试 什么样的错误处理代码才算是“更好的”？修改 `Mail_file` 的构造函数，处理与“-----”相关的可能的格式错误。

下面是 `find_from_addr()` 和 `find_subject()` 的实现，当前的版本只是简单地占据位置而已，并不是完整的实现，当我们能更好地从文件中识别信息(使用正则表达式，参见 23.6 ~ 23.10 节)时，将给出更好的实现方式，如下所示：

```
int is_prefix(const string& s, const string& p)
// is p the first part of s?
{
```

```

    int n = p.size();
    if (string(s,0,n)==p) return n;
    return 0;
}

bool find_from_addr(const Message* m, string& s)
{
    for(Line_iter p = m->begin(); p!=m->end(); ++p)
        if (int n = is_prefix(*p,"From: ")) {
            s = string(*p,n);
            return true;
        }
    return false;
}

string find_subject(const Message& m)
{
    for(Line_iter p = m.begin(); p!=m.end(); ++p)
        if (int n = is_prefix(*p,"Subject: ")) return string(*p,n);
    return "";
}

```

注意,我们使用子串的方式: `string(s, n)` 构造了一个包含 `s[n] ~ s[s.size() - 1]` 之间字符的子串,而 `string(s, 0, n)` 则构造了一个包含 `s[0] ~ s[n - 1]` 之间字符的子串。由于这些操作会创建新的字符串并进行字符复制,因此在使用上必须注意性能问题。

为什么 `find_from_addr()` 和 `find_subject()` 如此不同? 比如,一个返回 `bool` 值,而另一个返回 `string`。原因在于我们想说明以下两方面内容:

- `find_from_addr()` 应该区分有地址行但内容为空("")和无地址行两种不同的情况。对于第一种情况, `find_from_addr()` 返回 `true` (因为找到了地址行)并将 `s` 设置为空字符串"" (因为地址为空)。而对于第二种情况,应该返回 `false` (因为没有地址行)。
- 对于主题为空,或者没有主题行的情况, `find_subject()` 都返回""。

`find_from_addr()` 将两种情况区分开来,这是否有意义呢? 是否必要呢? 我们认为是有意义的,而且绝对是必要的。当在数据文件中查找信息时,会频繁出现这种不同情况间的细微差别: 我们是否找到了想要查找的域? 这个域中的内容是否有用? 在一个实际的程序中, `find_from_addr()` 和 `find_subject()` 都应该按照现在的 `find_from_addr()` 的风格来设计,以使用户能区分这种差别。

现在,这个版本的程序还没有进行过性能优化,但对于大多数应用场景来说,应该已经足够快了。特别是它对输入文件只读取一次,而且对于文件中的文本也不会保存多份拷贝。对于大文件,用 `unordered_multimap` 替换 `multimap` 可能会提高性能,但这未经过实验验证,无法下定论。

如果你对标准库中的关联容器(`map`、`multimap`、`set`、`unordered_map` 和 `unordered_multimap`) 还不太熟悉的话,请参考 21.6 节。

23.5 一个问题

I/O 流和 `string` 可以帮助我们读写、存储字符序列,并对其进行一些基本操作。但是,在应用中一个非常常见的问题是: 对于要处理的文本,我们需要考虑其中包含一个字符串或多个相似字符串的情况。我们来看一个很简单的例子: 查找一个电子邮件(一个单词序列)中是否包含美国某些州的邮政编码(州名的缩写加上邮政编码——两个字母后接 5 个数字),如下所示:

```

string s;
while (cin>>s) {

```

```

if (s.size()==7
    && isalpha(s[0]) && isalpha(s[1])
    && isdigit(s[2]) && isdigit(s[3]) && isdigit(s[4])
    && isdigit(s[5]) && isdigit(s[6]))
    cout << "found " << s << "\n";
}

```

其中 `isalpha(x)` 判断 `x` 是否是字母, `isdigit(x)` 判断 `x` 是否是数字(参见 11.6 节)。

这个简单(过于简单了)的解决方案存在若干问题:

- 它太冗长了(4 行代码, 8 个函数调用)。
- 我们忽略了所有没有用空白符将自身与上下文分开的邮政编码(如 "TX77845"、TX77845-1234 以及 ATX77845)。
- 我们忽略了(有意的?)所有用空白符分隔州名缩写和数字的邮政编码(如 TX-77845)。
- 我们将州名缩写为小写的字符串也识别为合法的邮政编码(有意的?)(如 tx77845)。
- 如果我们希望查找完全不同格式的邮政编码(例如 CB30FD), 不得不重写全部代码。

应该还有更好的解决方案! 但在设计新的方法之前, 我们来分析一下, 当处理更复杂的情况时, 如果还使用“简单有效的旧方法”来编写代码, 会遇到哪些问题:

- 如果希望处理多种格式, 我们不得不加入 `if` 语句和 `switch` 语句。
- 如果希望既能处理大写又能处理小写, 我们不得不进行显式的大小写转换(通常转换为小写)或者再加入 `if` 语句。
- 我们希望能以某种形式(何种形式?)来描述我们要查找的上下文。这意味着我们必须处理单个字符而不是字符串, 而且失去了 `iostream` 提供的很多优点(参见 7.8.2 节)。

如果你愿意, 可以使用旧方法实现上述功能。但显然, 沿着这条路走下去, 程序中会充斥着大量 `if` 语句, 来处理大量的特殊情况。即便是前面那个简单的例子, 我们也需要处理一些不同选项(例如, 同时支持 5 位和 9 位数字的编码)。对于其他很多实例, 我们都需要处理上下文中的重复(例如, 任意多个数字后接一个感叹号, 如 123! 和 123456!)。另外, 我们还必须处理前缀和后缀。就像 11.1 节和 11.2 节中讨论的那样, 人们对输出格式的偏好是会被程序员对规律性和简洁性的追求所局限的。你只要思考一下人们书写日期的五花八门的格式, 就会赞同这一观点:

```

2007-06-05
June 5, 2007
jun 5, 2007
5 June 2007
6/5/2007
5/6/07
...

```

此时, 甚至更早, 一个有经验的程序员就会下结论: “一定有更好的解决方法!”, 然后就去寻找新的方法了(而不会再用旧方法编写代码)。解决文本上下文搜索问题的一个最简单而且最通行的方法是使用所谓的正则表达式。正则表达式是大量文本处理的基础, UNIX 的 `grep` 命令就是基于正则表达式的(参见习题 8), 很多文本处理语言(如 AWK、PERL 以及 PHP)也都将支持正则表达式作为必备的功能。

我们借助一个库来实现基于正则表达式的搜索。这个库会成为下一版 C++ 标准(C++0x)的一部分, 它与 PERL 中的正则表达式处理程序是兼容的。这样, PERL 正则表达式处理的大量文档、教程和手册就都可以拿来作为参考。例如, 可以参考 C++ 标准委员会的工作文档(在互联网上搜索“WG21”即可找到), 或者是 John Maddok 写的 `boost::regex` 的文档, 以及大多数 PERL 的教程。在本章中, 我们会介绍正则表达式的一些基本概念和基本使用方法。

试一试 前面两个段落“粗心地”使用了几个你未见过的名词和缩写，而未加解释。请在互联网上搜索这些名词，了解它们的含义。

23.6 正则表达式的思想

正则表达式的基本思想是：定义一个模式，在文本中搜索这个模式。我们以邮政编码问题为例，看看对于 TX77845 这样的邮政编码如何定义模式。下面是第一个尝试：

`wwddddd`

其中 `w` 表示“任意字母”，`d` 表示“任意数字”。这里使用 `w` (表示“word”) 而不是 `l` (表示“letter”)，是因为 `l` 太容易与数字 `1` 混淆了。对于本例，这种符号表示是没有问题的，但我们还是来看一看它对于更复杂的情况，如 9 位数字的邮政编码格式 (TX77845-5629) 是否还有效：

`wwddddd-dddd`

这个模式看起来没有什么问题，但 `d` 如何就能表示“任意数字”，而“-”就表示它的字面意思——“连字符”吗？不管怎样，我们确实应该指出 `w` 和 `d` 具有特殊含义：它们表示字符集而非字符的字面含义 (`w` 表示“一个 `a` 或 `b` 或 `c` 或……”，`d` 表示“一个 `1` 或 `2` 或 `3` 或……”)。这有些过于微妙了，更好的表示方式是在这种表示一类单词的字母之前加上一个反斜线符号，以此来指出这是一个特殊符号，而不是字面含义，这与 C++ 语言中区分特殊符号的方式是相同的 (例如，`\n` 表示换行)。于是模式变为：

`\wwddddd-ddddd`

新的模式看起来有些丑陋，但至少不会引起歧义，而且反斜线符号显然具有一种“这是不寻常内容”的含义。在这里，我们表示一个字符多次重复的方式就是简单地重复几次。这样做令人厌烦，而且容易出错。我们真的在连字符之前获取了 5 个数字，而在其后获取了 4 个数字吗？答案是肯定的。但我们自始至终没有明确地表达 5 和 4，而是需要通过手工计数来保证数量正确。更好的方法是在字符之后用一个数值表示重复的次数，例如：

`\w2d5-\d4`

我们同样应该定义某种语法，来说明 2、5 和 4 是计数值，而不是表示文本中应该出现这几个数字。我们将计数值放在花括号中来表示这种含义：

`\w{2}\d{5}-\d{4}`

这种语法使得“`{}`”成为与“`\`”一样的特殊字符，但这是不可避免的。而且，当我们需要表示文本中出现花括号和反斜线符号的时候，也有办法处理。

到目前为止，看起来还不错，但我们还需要解决两个更为棘手的细节：最后 4 个数字是可选的。我们设计的模式，应该既能接受 TX77845，也能接受 TX77854-5629。解决这一问题有两种方式，一种是：

`\w{2}\d{5}或\w{2}\d{5}-\d{4}`

另一种是：

`\w{2}\d{5}和可选的-\d{4}`

为了能准确地描述这两种方式，我们首先需要有一种能表达分组 (子模式) 的语法，用来描述 `\w{2}\d{5}` 和 `-\d{4}` 是 `\w{2}\d{5}-\d{4}` 的两个组成部分。习惯上，我们用括号来表示分组的概念，例如：

`(\w{2}\d{5})(-\d{4})`

我们现在已经将模式划分为两个子模式了，接下来就可以描述我们最初的意图了——第二部分是可选的。与前面一样，新功能的引入伴随着新的特殊符号的引入，描述子模式的“(”现在与

“\”和“|”一样“特殊”了。而描述可选的概念，我们引入两个新的特殊符号：“|”用来表示“或”（两个子模式二选一）的概念，“?”用来表示某个子模式可选（有或无）的概念。于是，邮政编码后 4 位数字可选的第一种表示方式为：

```
(\w{2}\d{5})(\w{2}\d{5}-\d{4})
```

第二种方式：

```
(\w{2}\d{5})(-\d{4})?
```

与花括号表示计数一样（如 `\w{2}`），我们用问号（?）做后缀来表示可选的概念。例如，`(-\d{4})?` 表示“`-\d{4}` 是可选的”。也就是说，可以接受以一个连字符接 4 个数字为后缀的邮政编码；当然，没有这样后缀的编码也是可以接受的。实际上，5 位数字编码（`\w{2}\d{5}`）两边的括号没有任何作用，可以将其去掉：

```
\w{2}\d{5}(-\d{4})?
```

为了与 23.5 节中提出的需求完全吻合，我们在开始的两个字母之后加上一个可选的空格：

```
\w{2} \d{5}(-\d{4})?
```

“-”看起来有点怪，它实际就是一个空格后接一个?，表示空格是可选的。如果你不想用这么突兀的形式，可以把空格放在括号中，例如：

```
\w{2}(\ )?\d{5}((- \d{4})?)
```

如果还是觉得比较含糊，可以引入一个新的特殊符号来表示空格符，如 `\s`（`s` 表示“space”）。于是模式变为：

```
\w{2}\s?\d{5}(-\d{4})?
```

看起来已经达到最初的要求了，但是，如果有人在开头的两个字母之后写了两个空格，会发生什么情况呢？按现在的定义，模式会接受 `TX77845` 和 `TX 77845`，但不接受 `TX 77845`，这显然不符合要求。我们需要一种语法来描述“0 或多个空格符”，我们引入特殊符号“*”，以它作为后缀就表示“0 或多个”的含义。模式可以写为：

```
\w{2}\s*\d{5}(-\d{4})?
```

如果你随着本节的讲述一步步地走过来，会觉得这个最终的正则表达式很合理。这种表示模式的方法符合逻辑而且非常简洁。而且，我们并非随意地选择了一些描述方法，本节所介绍的符号表示都是非常普遍和通用的。对于很多文本处理工作，你必须编写、阅读这种符号表示。当然，这种描述方法看起来有些古怪，好像是你家的小猫在键盘上散步所产生符号串。而且，采用这样的描述方法，敲错任何一个符号（甚至是一个空格）都可能完全改变其含义。但是，请尽量熟悉它。我们无法找到任何其他描述方法能明显优于正则表达式。而且，这种描述风格自 30 年前由 UNIX 的 `grep` 命令引入后，一直流行到现在，甚至从未进行过大的修改。

23.7 用正则表达式进行搜索

现在，我们可以使用上一节定义的邮政编码的模式来搜索文件中的邮政编码了。程序先定义模式，然后逐行读取文件，在其中搜索模式。如果在某行中找到了模式，则输出行号，如下所示：

```
#include <boost/regex.hpp>
#include <iostream>
#include <string>
#include <fstream>
using namespace std;

int main()
{
```

```

ifstream in("file.txt"); // input file
if (!in) cerr << "no file\n";

boost::regex pat ("\\w{2}\\s*\\d{5}(-\\d{4})?"); // ZIP code pattern
cout << "pattern: " << pat << '\n';

int lineno = 0;
string line; // input buffer
while (getline(in, line)) {
    ++lineno;
    boost::smatch matches; // matched strings go here
    if (boost::regex_search(line, matches, pat))
        cout << lineno << ": " << matches[0] << '\n';
}

```

程序的一些细节需要解释一下，首先看看下面的程序：

```

#include <boost/regex.hpp>
...
boost::regex pat ("\\w{2}\\s*\\d{5}(-\\d{4})?"); // ZIP code pattern
boost::smatch matches; // matched strings go here
if (boost::regex_search(line, matches, pat))

```

我们这里使用的是 boost 中的 regex 库，它很快就会成为 C++ 标准库的一部分了。在使用之前，你需要安装这个库。我们使用显式的限定符 `boost::regex` 指明所使用的函数和类型来自于 regex 库。

我们再回到正则表达式，看这部分代码：

```

boost::regex pat ("\\w{2}\\s*\\d{5}(-\\d{4})?");
cout << "pattern: " << pat << '\n';

```

此处我们先定义了一个模式 `pat` (类型为 `regex`)，并将其输出到屏幕。注意，我们定义 `pat` 为：

```
"\\w{2}\\s*\\d{5}(-\\d{4})?"
```

运行程序，输出的结果是：

```
pattern: \w{2}\s*\d{5}(-\d{4})?
```

我们知道，在 C++ 字符串中，反斜线是转义字符(参见附录 A.2.4)。因此要得到反斜线符号本身的话，需要写两个反斜线“\\”。

一个 regex 模式实际上也是一个字符串，因此我们可以用“<<”输出其内容。但一个 regex 不仅仅是一个字符串，它还是一种复杂的模式匹配机制。当初初始化一个 regex 变量时，这个机制就建立起来了。这种复杂机制已经超出了本书的讨论范围，但我们无需了解这些，我们只要知道，一旦用上节定义的模式初始化了一个 regex 变量，我们就可以用它在文件的每一行中搜索邮政编码了：

```

boost::smatch matches;
if (boost::regex_search(line, matches, pat))
    cout << lineno << ": " << matches[0] << '\n';

```

`regex_search(line, matches, pat)` 搜索 `line` 中与正则表达式 `pat` 匹配的内容，如果找到，则将结果保存在 `matches` 中。如果未找到匹配内容，则返回 `false`。

变量 `matches` 的类型是 `smatch`，前缀 `s` 表示“子(匹配)”(`sub`)的概念。一个 `smatch` 本质上是一个子匹配的向量。第一个元素(本例中为 `matches[0]`)是完整匹配。如果 `i < matches.size()`，我们可以将 `matches[i]` 当做一个字符串。对于一个正则表达式，如果最多有 `N` 个子模式，则 `matches.size() == N + 1`。

那什么是子模式呢？一个较好的初步的回答是：“模式中任何放在括号中的内容都可以作为

一个子模式。”如模式“`\\w{2}\\s*\\d{5}(-\\d{4})?`”中,我们唯一能看到的子模式是 4 位扩展数字,因为它是在括号中的,因此我们猜测(实际就是这样)`matches.size() == 2`。这样,我们猜测可以通过 `matches` 很容易地访问后 4 位数字,如下面代码所示:

```
while (getline(in,line)) {
    boost::smatch matches;
    if (boost::regex_search(line, matches, pat)) {
        cout << lineno << ": " << matches[0] << '\n';    // whole match
        if (1 < matches.size() && matches[1].matched)
            cout << "\t: " << matches[1] << '\n';    // sub-match
    }
}
```

严格来说,我们不必测试 `1 < matches.size()`,因为我们已经知道模式的详细结构。但最好还是加上这个检测(虽然这令我们看起来有点像偏执狂),因为我们已经试验过多种不同的模式,并非所有模式都恰好有一个子模式。我们可以通过 `matches` 中的(对位元素),来判断一个子模式是否匹配成功。在本例中,是通过 `matches[1].matched` 来判断的:当 `matches[i].matched` 为假时,即子模式未匹配时,`matches[i]`的内容会是一个空字符串。类似地,不存在的子模式(如对本例的模式访问 `matches[17]`)会按未匹配来处理。

对包含如下内容的文件测试我们的程序:

```
address TX77845
ffff tx 77843 asasasaa
ggg TX3456-23456
howdy
zzz TX23456-3456sss ggg TX33456-1234
cvzcv TX77845-1234 sdsas
xxxTx77845xxx
TX12345-123456
```

得到如下输出结果:

```
pattern: "\\w(2)\\s*\\d(5)(-\\d(4))?"
1: TX77845
2: tx 77843
5: TX23456-3456
: -3456
6: TX77845-1234
: -1234
7: Tx77845
8: TX12345-1234
: -1234
```

注意:

- 我们未被 `ggg` 那行中错误的格式所欺骗。(它错在哪里?)
- 在 `zzz` 那行中,我们只找到了第一个邮政编码(本来就是要求每行只找一个)。
- 在第 5 行和第 6 行中我们正确地找到了后缀形式的编码。
- 我们找到了第 7 行中“隐藏”在 `xxx` 中的编码。
- 我们找到了隐藏在 `TX12345-123456` 中的编码(`TX12345-1234`,这样做是否不正确?)。

23.8 正则表达式语法

上一节介绍了一个较为简单的正则表达式匹配的例子。下面我们更为系统、完整地介绍一下正则表达式(以 `regex` 库为线索来介绍)。

正则表达式(简称正则式,“`regex`”或“`regex`”)实际上是一种表达字符模式的语言,只不过这

种语言的规模很小。它是一种强大(表达能力强)而简洁的语言,而又有些神秘。经过几十年的使用,产生了很多微妙的特性和“方言”。在本章中,我们只介绍它的一个子集,这也是使用最为广泛的一种方言(PERL)。如果你希望了解更多的特性以便表达更复杂的模式,或者你希望了解其他方言,请搜索互联网。网络上相关的学习指南(质量差异很大)俯拾即是。特别是,boost::regex 规范及其标准委员会文档(WG21 TR1)很容易找到。

boost regex 库还支持 ECMAscript、POSIX、awk、grep 和 egrep 表示法和许多搜索选项。这是非常有用的,特别是当你需要使用的正则式是用其他语言设计的时候。如果你需要了解这些额外的特性,可以查阅相关资料。不过,请记住,“使用最多的特性”不是一个好的程序设计风格。无论什么时候,都请替可怜的程序维护人员着想(很有可能就是你自己),他需要阅读并理解你的代码:因此编写代码时不要炫耀你的聪明,并且避免使用那些晦涩难懂的特性。

23.8.1 字符和特殊字符

一个正则式描述了一个模式,用来在字符串中查找匹配的字符。默认情况下,模式中的一个字符在字符串中就匹配它自身。例如,正则式“abc”就匹配“Is there an abc here?”中的 abc。

正则式的强大来自于具有特殊含义的“特殊字符”以及字符组合:

特殊含义的字符	
.	任意单个字符(通配符)
[字符集
{	计数
(子模式开始
)	子模式结束
\	下一个字符具有特殊含义
*	0 个或多个
+	一个或多个
?	可选(0 个或一个)
	二选一(或)
^	行的开始;否定
\$	行的结束

例如:

x.y

匹配任何以 x 开头并且以 y 结束的的长度为 3 的字符串,例如 xxy、x3y 和 xay,但不匹配 yxy, 3xy 及 xy。

注意,{...}、*、+ 以及? 是后缀运算符。例如,\d+表示“一个或多个十进制数字”。如果你想在模式中使用这些特殊符号的普通字符含义,需要利用反斜线进行“转义”。例如,+表示“一个或多个”运算符,而\+表示加号。

23.8.2 字符集

最常用的字符组合通常也用特殊字符的简洁形式表示:

表示字符集的特殊字符		
\d	一个十进制数字	[[[:digit:]]]
\l	一个小写字母	[[[:lower:]]]

(续)

表示字符集的特殊字符		
<code>\s</code>	一个空白符(空格符、制表符等)	<code>[:space:]</code>
<code>\u</code>	一个大写字母	<code>[:upper:]</code>
<code>\w</code>	一个字母(a~z 或 A~Z)或数字(0~9)或下划线(_)	<code>[:alnum:]</code>
<code>\D</code>	除了\ d 之外的字符	<code>^[^digit:]</code>
<code>\L</code>	除了\ l 之外的字符	<code>^[^lower:]</code>
<code>\S</code>	除了\ s 之外的字符	<code>^[^space:]</code>
<code>\U</code>	除了\ u 之外的字符	<code>^[^upper:]</code>
<code>\W</code>	除了\ w 之外的字符	<code>^[^alnum:]</code>

注意, 大写形式的特殊字符表示“除了对应的小写形式特殊字符所表示的字符之外的所有字符”。特别地, `\W` 表示“不是一个字母”而非“一个大写字母”。

第三列的内容(如`[:digit:]`)是表示相同含义的另一种较长的表示方式。

与 `string` 和 `iostream` 库类似, `regex` 库可以处理大字符集, 如 `Unicode`。与前文一样, 我们不对此进行详细介绍, 需要时你可以查找相关资料或求助于有经验的人。`Unicode` 文本处理已经超出了本书的讨论范围。

23.8.3 重复

模式的重复通过一些后缀运算符来实现:

重复	
<code>{n}</code>	严格重复 n 次
<code>{n,}</code>	重复 n 次或更多次
<code>{n,m}</code>	重复至少 n 次,至多 m 次
<code>*</code>	重复 0 次或多次,即{0,}
<code>+</code>	重复一次或多次,即{1,}
<code>?</code>	可选(0 次或一次),即{0,1}

例如:

`Ax*`
与任何以 A 开始, 后接 0 或多个 x 的字符串匹配, 例如:
`A`
`Ax`
`Axx`
`AAAAAAAAAAAAAAAAAAAAAAAAAAAA`

如果希望字符至少出现一次, 则用 `+` 替换 `*`。例如:

`Ax+`
匹配那些以 A 开始, 后接一个或多个 x 的字符串, 如
`Ax`
`Axx`
`AAAAAAAAAAAAAAAAAAAAAAAAAAAA`

但不匹配

`A`
常用的出现 0 次或一次(“可选的”)的概念用问号表示。例如:

`\d-?d`

匹配以破折号分隔的两个数字和连续两个数字, 例如:

`1-2`

`12`

但不匹配

`1--2`

如需指定特定的重复次数, 或者一定范围内的重复次数, 可用花括号。例如:

`\w{2}-d{4,5}`

匹配以两个字母(或数字、下划线)和连字符开始, 后接 4 个或 5 个数字的字符串, 如

`Ab-1234`

`XX-54321`

`22-54321`

但不匹配

`Ab-123`

`?b-1234`

注意, 数字也属于字符集 `\w`。

23.8.4 子模式

为了指定模式中的子模式, 用括号将其括起来。例如:

`(d*?)`

它定义了一个子模式, 表示 0 或多个数字后接一个冒号。一个复杂的模式可用多个子模式组成。例如:

`(d*?)(d+)`

它表示字符串前半部分可以为空, 若非空, 则是任意长度的数字序列(可以为空)后接一个冒号, 后半部分是一个或多个数字的序列。多么冗长的叙述! 难怪人们发明正则表达式这样一种简洁、准确的方法来描述这些模式。

23.8.5 可选项

“或”运算符(`|`)表示二选一的概念。例如:

`Subject: (FW:|Re:)?(.*)`

匹配主题行, 其中包含可选的 `FW:` 或 `Re:`, 后接 0 个或多个任意字符。例如, 它匹配如下字符串:

`Subject: FW: Hello, world!`

`Subject: Re:`

`Subject: Norwegian Blue`

但不匹配:

`SUBJECT: Re: Parrots`

`Subject FW: No subject!`

注意, 或运算的两个子正则式均不能为空:

`(|def) // error`

多个连续的或运算是允许的:

`(bs|Bs|bS|BS)`

23.8.6 字符集和范围

前文已经介绍了一些表示字符集的特殊字符, 如表示数字的 `\d`, 表示字母、数字或下划线的 `\w` 等(参见 23.7.2 节)。不过, 有时我们还需要定义其他的字符集, 这也很容易, 例如:

`[\w @]`

字母、数字、下划线、空格或 `@`

`[a-z]`

小写字母 `a~z`

[a-zA-Z]	大写或小写字母 a~z(或 A~Z)
[Pp]	大写或小写的 p
[\w-]	字母、数字、下划线或破折号
[asdfghjkl;']	美式 QWERTY 键盘中间一行上的所有字符
[.]	句点
[.{(\ *+?^\$}]	所有特殊字符(这里表示字符本身,不是特殊字符的含义)

在字符集中, -(连字符)表示范围,如[1-3]表示1、2或3,[w-z]表示w、x、y或z。范围的使用一定要很小心:并非所有语言都具有相同的字母,而且并非所有字符集中字符顺序都一致。如果你觉得要使用的范围不是最常见的英语字母表中的字母或者数字范围,请查阅相关资料。

注意,我们可以在字符集中使用特殊字符,如\w。于是产生一个问题,如何表示反斜线符号?与往常一样,对其进行转义即可:"\\".

如果字符集的第一个字符是^,则表示“非”的概念。例如:

[^aeiouy]	非英语元音
[^\\d]	非数字
[^aeiouy]	英语元音或空格

在最后一个正则式中,^不是字符集中的第一个字符,因此它只是一个普通字符,而不是非运算符——正则表达式就是如此微妙。

regex 的一些实现中还提供了一组命名字符集。例如,如果希望匹配字母数字符号(即匹配一个字母或者一个数字:a-z或A-Z或0-9),可以使用[[:alnum:]]。在这里,alnum 是字符集的名称(字母数字字符集)。于是,加引号的非空字母数字串对应的正则式为"[[:alnum:]]+"。为了将此正则式放在程序中的字符串内,需要将引号转义:

```
string s = "\\[[:alnum:]]+\\\"";
```

而且,在 regex 中引号也是特殊符号。因此为了将字符串转换为 regex 对象,我们还需再产生一个反斜线符号,使得在 regex 对象中引号被转义,而不是表示特殊符号。另外,我们需要使用()风格初始化 regex,因为利用 string 构造 regex 必须是显式的:

```
regex s("\\\\[[:alnum:]]+\\\\\\");
```

下表列出了一些标准的命名字符集:

字符集	
alnum	任意字母数字
alpha	任意字母
blank	任意空白符,不包括换行
cntrl	任意控制字符
d	任意十进制数字
digit	任意十进制数字
graph	任意图形字符
lower	任意小写字母
print	任何可打印字符

(续)

字符集	
punct	任意标点字符
s	任意空白符
space	任意空白符
upper	任意大写字母
w	任意单词字符(字母、数字及下划线)
xdigit	任意十六进制数字字符

特定的 regex 实现可能提供更多的命名字符集,但如果你决定使用的字符集不在上表内,一定要检查可移植性是否足够好,是否能满足你最初的需求。

23.8.7 正则表达式错误

如果你指定了一个错误的正则表达式,会产生什么后果?例如:

```
regex pat1("(lghi)");    // missing alternative
regex pat2("[c-a]");     // not a range
```

当我们将一个模式赋予 regex 时,它会对模式进行检查,如果发现模式不合法或者过于复杂,无法用于匹配时,它会抛出一个 bad_expression 异常。

下面这段程序对体会正则表达式匹配很有帮助:

```
#include <boost/regex.hpp>
#include <iostream>
#include <string>
#include <fstream>
#include <sstream>
using namespace std;
using namespace boost;    // if you use the boost implementation

// accept a pattern and a set of lines from input
// check the pattern and search for lines with that pattern

int main()
{
    regex pattern;

    string pat;
    cout << "enter pattern: ";
    getline(cin,pat);      // read pattern

    try {
        pattern = pat;    // this checks pat
        cout << "pattern: " << pattern << '\n';
    }
    catch (bad_expression) {
        cout << pat << " is not a valid regular expression\n";
        exit(1);
    }

    cout << "now enter lines:\n";
    string line;           // input buffer
    int lineno = 0;

    while (getline(cin,line)) {
```



```

    ++lineno;
    smatch matches;
    if (regex_search(line, matches, pattern)) {
        cout << "line " << lineno << ": " << line << '\n';
        for (int i = 0; i<matches.size(); ++i)
            cout << "\tmatches[" << i << "]: "
                << matches[i] << '\n';
    }
    else
        cout << "didn't match\n";
}
}
```

试一试 编译、运行这个程序，尝试一些模式，如 `abc`、`x.*x`、`(.*)`、`\([^\)]*\)` 以及 `\w+ \w+ (Jr\.)?`。

23.9 与正则表达式进行模式匹配

正则表达式有两种主要用途：

- 在(任意长的)数据流中搜索与正则式匹配的字符串——`regex_search()` 就可以实现此功能，它在数据流中搜索与正则式匹配的子串。
- 判断一个字符串(已知长度)是否与模式匹配——`regex_match()` 检查模式和给定字符串是否完全匹配。

23.6 节中给出的搜索 ZIP 的程序是正则式搜索功能的很好示例。下面，我们介绍一个匹配操作的例子。例如，我们需要从像下表这样的结构中提取数据：

KLASSE	ANTAL DRENGE	ANTAL PIGER	ELEVER IALT
0A	12	11	23
1A	7	8	15
1B	4	11	15
2A	10	13	23
3A	10	12	22
4A	7	7	14
4B	10	5	15
5A	19	8	27
6A	10	9	19
6B	9	10	19
7A	7	19	26
7G	3	5	8
7I	7	3	10
8A	10	16	26
9A	12	15	27
OMO	3	2	5
OP1	1	1	2
OP2	0	5	5
10B	4	4	8

(续)

KLASSE	ANTAL DRENGE	ANTAL PIGER	ELEVER IALT
10CE	0	1	1
1MO	8	5	13
2CE	8	5	13
3DCE	3	3	6
4MO	4	1	5
6CE	3	4	7
8CE	4	4	8
9CE	4	9	13
REST	5	6	11
Alle klasser	184	202	386

这个表记录的是 Bjarne Stroustrup 的母校在 2007 年的学生数，它实际上是从互联网上提取出来的，其原始格式看起来很整洁，而且正是我们进行数据分析时常见的那种典型格式：

- 它包含数值域。
- 它包含字符域，其中字符串只有了解上下文的人才知道其含义。（在本例中，这种情况更为明显，因为文字都是丹麦文。）
- 字符串中包含空格。
- 数据“域”用“分隔符”分开，在本例中，分隔符为制表符。

我们选择的这个例子“相当典型”而且“不是很困难”，但有一点比较微妙：人眼是无法看出空格和制表符之间的区别的，这只能在程序中进行区分。

我们将展示正则表达式的如下用途：

- 验证表格布局是否正确（即是否每行包含的域的数目都正确）。
- 验证合计值是否正确（每列最后一行上的数值为其上所有数值之和）。

如果我们可以完成这些任务，那么我们就几乎能做任何事！例如，由原表格创建出一个新的表格：将具有相同起始数字的行（表示年级：一年级用 1 表示，依此类推）合并在一起，或者分析学生数是逐年增长还是减少（参考习题 10 和 11）。

为了对表格进行分析，我们需要两个模式：一个用于分析表头行，另一个用于分析剩余行：

```
regex header( "^\\w ]+( \\w ]+)*$");
regex row( "^\\w ]+( \\d+)( \\d+)( \\d+)$");
```

请记住，我们一直在称赞正则表达式简洁、功能强大，但我们从未称赞它易于被初学者理解。实际上，“只写语言”的名声对正则式来说是恰如其分的。我们从表头开始，由于它（第一行）不包含任何数值，将其直接丢弃即可。但是，为了多做一些练习，我们还是对其进行分析。它包含 4 个由制表符分隔的“单词域”（“字母数字域”）。这些域中可以包含空格，因此，我们不能简单地用 `\w` 来匹配其中的字符，而应该用 `[\w]`，即一个字母、数字、下划线或者一个空格。因此，匹配第一个域的正则式为 `[\w] +`。我们希望第一个域在行首，因此可用 `^([\w] +)` 匹配，符号“`^`”表示“行首”的含义。行中剩余域可描述为一个制表符后接多个单词：`([\w] +)`。现在，我们先给出匹配任意多个这种域，最后是行尾的正则式：`([\w] +) * $`。美元符号（`$`）表示“行尾”。我们将完整的正则式写成 C++ 字符串的形式，如前所述，需要添加一些额外的反斜线符号：

```
"^[\\w ]+( [\\w ]+)*$"
```

注意，人眼是看不出制表符和空格之间的区别的，但在本例中，排版时已经将制表符展开了，因此可以明确地区分开来。

现在来看更有趣的部分：如何为数值行设计模式。如表头行一样，数值行的行首也是单词域，因此子正则式为`^[\\w]+`。后面是三个数值域，每个域之前是一个制表符：`(\\d+)`，因此，完整的正则式为：

```
^[\\w ]+( \\d+)( \\d+)( \\d+)$
```

放入 C++ 字符串：

```
"^[\\w ]+( \\d+)( \\d+)( \\d+)$"
```

现在，模式已经设计完毕，下面所要做的就是使用它们分析表格。首先验证表格布局：

```
int main()
{
    ifstream in("table.txt");    // input file
    if (!in) error("no input file\n");

    string line;    // input buffer
    int lineno = 0;

    regex header( "^[\\w ]+( [\\w ]+)*$");    // header line
    regex row( "^[\\w ]+( \\d+)( \\d+)( \\d+)$");    // data line

    if (getline(in,line)) {    // check header line
        smatch matches;
        if (!regex_match(line, matches, header))
            error("no header");
    }
    while (getline(in,line)) {    // check data line
        ++lineno;
        smatch matches;
        if (!regex_match(line, matches, row))
            error("bad line",to_string(lineno));
    }
}
```

简洁起见，我们省略了`#include`。我们的目的是检查每行中的所有字符，因此我们使用`regex_match()`而不是`regex_search()`。两者的区别在于，`regex_match()`需匹配输入中所有字符才能判断匹配成功，而`regex_search()`只要在输入中找到匹配的字串即可。如果你想用的是`regex_search()`，但误输入了`regex_match()`（或反之），这种错误将很难查找出来。不过，两个函数对参数的使用是相同的。

接下来我们对表中的数据进行验证。我们对男孩（“dreng”）和女孩（“piger”）两列保存其学生数之和。对每一行，我们检查最后一个域（“ELEVER IALT”）是否等于前两个域之和。最后一行（“Alle klasser”）的内容是同列中其他数据的合计值。为了进行这些检查，我们修改了模式`row`，将文本域设计为子模式，这样就可以识别“Alle klasser”了，例如：

```
int main()
{
    ifstream in("table.txt");    // input file
    if (!in) error("no input file");

    string line;    // input buffer
    int lineno = 0;

    regex header( "^[\\w ]+( [\\w ]+)*$");
```

```

regex row( "^(\\w+)(\\d+)(\\d+)(\\d+)$");

if (getline(in,line)) { // check header line
    boost::smatch matches;
    if (!boost::regex_match(line, matches, header)) {
        error("no header");
    }
}

// column totals:
int boys = 0;
int girls = 0;

while (getline(in,line)) {
    ++lineno;
    smatch matches;
    if (!regex_match(line, matches, row))
        cerr << "bad line: " << lineno << "\n";

    if (in.eof()) cout << "at eof\n";

    // check row:
    int curr_boy = from_string<int>(matches[2]);
    int curr_girl = from_string<int>(matches[3]);
    int curr_total = from_string<int>(matches[4]);
    if (curr_boy+curr_girl != curr_total) error("bad row sum\n");

    if (matches[1]=="Alle klasser") { // last line
        if (curr_boy != boys) error("boys don't add up\n");
        if (curr_girl != girls) error("girls don't add up\n");
        if (!(in>>ws).eof()) error("characters after total line");
        return 0;
    }
    // update totals:
    boys += curr_boy;
    girls += curr_girl;
}

error("didn't find total line");
}

```

最后一行在语义上与其他行是不同的——它是其他行之和，我们通过标签“Alle klasser”来识别它。在这一行之后，我们不再接受任何非空白字符（使用来自 `lexical_cast()` 的技术，参见 23.2 节），如果未找到这一行（合计值），则输出一个错误信息。

我们使用 23.2 节中的 `from_string()` 从数据域中提取整数值。我们已经确认这些域中只包含数字，因此无需检查这次字符串到整数的转换是否成功。

23.10 参考文献

正则表达式是一种很流行，也很有用的工具。很多程序设计语言都支持正则表达式，其格式也各种各样。其理论基础是一种优美的形式语言理论，其高效的实现技术则基于状态机。正则表达式的全部概念、基础理论、实现以及状态机的一般用法已经超出了本书的讨论范围。不过，由于这些主题都是计算机科学课程中重要的内容，而正则式又如此流行，因此如果你需要学习这些内容或者仅仅是感兴趣的话，很容易找到相关资料。一些参考文献罗列如下：

- Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools, Second Edition* (usually called “The Dragon Book”). Addison-Wesley, 2007. ISBN 0321547985.
- Austern, Matt, ed. “Draft Technical Report on C++ Library Extensions.” ISO/IEC DTR 19768, 2005 www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1836.pdf.

Boost.org. A repository for libraries meant to work well with the C++ standard library. www.boost.org.
 Cox, Russ. "Regular Expression Matching Can Be Simple and Fast (but Is Slow in Java, Perl, PHP, Python, Ruby, ...)." <http://swtch.com/~rsc/regex/regexpl.html>.
 Maddoc, J. boost::regex documentation. www.boost.org/libs/regex/doc/index.html.
 Schwartz, Randal L., Tom Phoenix, and Brian D. Foy. *Learning Perl, Fourth Edition*. O'Reilly, 2005. ISBN 0596-101058.

简单练习

1. 确认你的机器上安装的标准库是否包含 `regex`。提示：尝试使用 `std::regex` 和 `tr1::regex`。
2. 编译、运行 23.7.7 节中的小程序，可能需要安装 `boost::regex`，并弄清如何通过设置工程属性或命令行选项来使用 `regex` 头文件及链接 `regex` 库。
3. 使用上一小题中的程序测试 23.7 节中的模式。

思考题

1. 我们在哪里查找“文本”？
2. 标准库中哪些功能对于文本分析非常有用？
3. `insert()` 的插入位置是其位置（或迭代器）之前还是之后？
4. Unicode 是什么？
5. 如何将字符串转换为其他类型？反过来呢？
6. 假定 `s` 是一个字符串，`cin >> s` 和 `getline(cin, s)` 的区别在哪里？
7. 列出标准流。
8. 一个映射对象中的关键字是什么？给出一些关键字类型的例子。
9. 如何遍历映射的元素？
10. `map` 和 `multimap` 的差别在哪？哪种有用的 `map` 的操作在 `multimap` 中不存在，这样设计的原因是什么？
11. 向前迭代器需要哪些操作？
12. 空域和域不存在有什么区别？给出两个例子。
13. 正则表达式中为什么需要使用转义符？
14. 如何将正则表达式存入 `regex` 变量？
15. `\w + \s\d|4|` 与什么样的字符串匹配？给出三个例子。如果将此模式转换为 `regex` 变量，需要用什么样的字符串初始化 `regex` 变量？
16. 在程序中，如何确定一个字符串是否是合法的正则表达式？
17. `regex_search()` 的功能是什么？
18. `regex_match()` 的功能是什么？
19. 如何在正则表达式中表示句点符号（.）？
20. 如何在正则表达式中表示“至少三个”的概念？
21. 字符 `7` 与 `\w` 匹配吗？下划线符号 `_` 呢？
22. 如何在正则表达式中表示大写字母？
23. 如何自定义字符集？
24. 如何从整数域中提取数值？
25. 如何用正则表达式表示浮点数？
26. 如何从匹配结果中提取浮点值？
27. 子匹配是什么？如何访问子匹配结果？

术语

匹配	<code>regex_match()</code>	搜索
<code>multimap</code>	<code>regex_search()</code>	<code>smatch</code>
模式	正则表达式	子模式

习题

1. 编译、运行邮件文件分析程序，创建一个更大的邮件文件来测试它。一定要加入一些可能触发错误的邮件消息，例如有两个地址行的邮件、多个邮件具有相同的地址和/或相同的主题、空邮件等。另外，用一些显然不符合程序定义的邮件形式的内容进行测试，例如，一个不包含“----”行的大文件。
2. 添加一个 `multimap` 对象，用来保存主题。修改程序，令其从键盘接收一个字符串，输出所有主题与此字符串匹配的邮件。
3. 修改 23.4 节中的邮件分析程序，使用正则表达式查找主题和发件人。
4. 找一个真正的邮件文件(包含真实邮件消息)，修改邮件分析程序，使其能提取指定发件人的邮件的主题行。
5. 找一个大的邮件文件(包含几千个邮件消息)，测试用 `multimap` 输出所有邮件消息所花费的时间，测试改用 `unordered_multimap` 后的时间。注意，我们的应用并未利用 `multimap` 的优点。
6. 编写一个程序，从一个文本文件中查找日期。输出包含日期的行，格式为“行号: 行内容”。以一个简单的日期格式为起点，如 12/24/2000，设计、测试程序。随后再加入更多的格式。
7. 编写程序(与上题类似的程序)，在文件中查找信用卡号码。上网搜索一下真实的信用卡号码是什么格式。
8. 修改 23.8.7 节中的程序，使其接受一个模式和一个文件名作为输入，输出文件中匹配模式的行，输出格式为“行号: 行内容”。如果未找到匹配行，则不输出任何内容。
9. 使用 `eof()`(参见附录 B.7.2)来检测某行是否是表格的最后一行。采用这种方法简化 23.9 节中的程序。用表格后接空行的文件和不以换行结束的文件测试程序。
10. 修改 23.9 节中的表格验证程序，用原表格中的数据创建并输出一个新表格，其中所有首数字相同(同一年级)的行被合并在一起。
11. 修改 23.9 节中的表格验证程序，检查学生数是逐年增加还是减少。
12. 基于习题 6 中的程序，编写一个新程序，查找所有日期并将格式改为 ISO 标准格式 `yyyy/mm/dd`。程序读入输入文件，转换日期格式后将结果写入输出文件。两个文件内容完全一致，只是日期格式可能不同。
13. 句点(.)与'\n'匹配吗？编写一个程序验证一下。
14. 编写一个程序，类似 23.8.7 中的程序，可以输入模式进行匹配。但是，它从文件(以'\n'作为行的分隔)读取输入，因此可以测试跨行的模式。测试这个程序，并记录至少一打以上的测试结果。
15. 给出一个不能用正则表达式描述的模式。
16. 本题不适合初学者：证明上题中的模式确实不是正则表达式。

附言

我们很容易陷入这样一个观点：计算机和计算都是面对数字的，计算就是数学的一种形式。这显然是不正确的。只要看看你的计算机屏幕就很清楚了，上面充满了文本和图片。甚至说不定它正在播放音乐呢。对于不同应用，使用适当的工具是非常重要的——从 C++ 的角度，就是要使用适合的库。对于文本处理，正则表达式库通常是关键工具——另外不要忘了映射和标准库算法。

第24章 数值计算

“每个复杂问题都存在一个清晰、简洁但是错误的解答。”

——H. L. Mencken

本章介绍用于数值计算的一些基本语言特性和标准库功能。我们提出大小、精度以及截断等一些基本问题。本章的核心部分是关于多维数值的讨论，既讨论 C 风格的多维数组，也介绍 N 维矩阵库。我们还将介绍随机数，它被广泛用于测试、仿真以及电脑游戏中。最后，我们介绍标准库数学函数，并简要介绍标准库对复数的支持。

24.1 介绍

对某些人来说，数字、数值计算就是一切，比如很多科学家、工程师以及统计学家等。对更多的人来说，数值计算在某些时候是必要的。例如，一个计算机科学家偶尔与一个物理学家合作时，就属于这种情况。而对于大多数人来说，很少会用到数值计算（不是整数和浮点数的简单算术运算，而是更复杂的计算）。本章的目的是介绍一些用于处理简单数值计算问题的程序设计语言技术细节。我们不会介绍数值分析或者浮点数运算的微妙难懂之处，这些内容已经远远超出了本书的讨论范围，而且与应用中领域相关的问题是紧密融合的。本章主要讨论如下问题：

- 一些内置类型是有固定大小，由此引发的精度、溢出等问题。
- 数组：内置的多维数组类型和更适于数值计算的 Matrix 库。
- 随机数的最基本的概念。
- 标准库中的数学函数。
- 复数。

其中 Matrix 库是重点，它使矩阵（多维数组）的处理变得简单。

24.2 大小、精度和溢出

当我们使用内置类型和普通计算技术时，数值会占用固定大小的内存。也就是说，整数类型（int、long 等）只是数学上的整数的近似。同样，浮点数类型（float、double 等）也只是数学上的实数的近似。这意味着，从数学的角度看，某些计算机中的计算是不精确的，甚至是错的，例如：

```
float x = 1.0/333;  
float sum = 0;  
for (int i=0; i<333; ++i) sum+=x;  
cout << setprecision(15) << sum << "\n";
```

执行这段代码，一些人很天真地期望得到 1，但实际得到的结果是

0.999999463558197

这就是我们所期待的结果，因为我们了解计算机数值计算——这就是一个截断误差的例子。在计算机中，一个浮点数只占用固定数目的二进制位，因此我们总是可以“欺骗”计算机：让它做一个运算，其结果需要更多的二进制位来保存。例如，有理数 $1/3$ 是无法用十进制数精确表示的（无

论我们使用多少位十进制数字都不能)。1/333 也是如此,于是,当我们将 x (计算机中与 1/333 最为接近的浮点数值)累加 333 次时,我们得到是与 1 有微小差距的一个值。每当我们进行大量浮点数运算时,就会产生截断误差,唯一的问题是误差对结果的影响是否严重。

要时时检查计算结果是否合理。当进行计算时,你必须清楚什么是“合理的结果”,否则就很容易被“愚蠢的错误”或者计算误差所愚弄。要保持对截断误差的警惕,如果有疑问,一定要请教专家或者仔细研究数值计算的相关资料。

试一试 将上例中的 333 改为 10,重新运行程序。你预计会得到什么结果?实际得到了什么结果?我们早已警告过你了!

相对于实数,用固定位数表示整数所引起的问题更为引人注目。原因在于,浮点数被定义为实数的近似,因此后果是丢失精度(即丢失最低有效位),而整数则是引起溢出(即丢失最高有效位)。因此,浮点数运算的误差总是比较细微,不易被初学者察觉,而整数的误差则往往非常惊人,很难不被注意。请记住,我们希望错误更早地、更突出地显现出来,以便能及时修正。

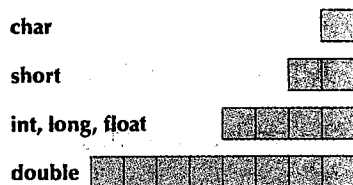
看看下面的程序:

```
short int y = 40000;
int i = 1000000;
cout << y << " " << i*i << "\n";
```

其输出结果为:

```
-25536    -727379968
```

这就是典型的溢出现象。我们当然希望能表示任意整数值,以获得准确的计算结果。但计算机中的整型只能表示(相对)较小的整数,其宽度不足以精确表示所有整数。在本例中,一个两个字节的 short 类型不能表示 40 000,而一个 4 个字节的 int 类型不能表示 1 000 000 000 000。C++ 内置类型的准确宽度依赖于硬件平台和编译器(参考附录 A.8)。我们可以使用 `sizeof(x)` 来获得 x 的宽度(以字节为单位), x 可以是一个变量或者一个类型。由定义得到 `sizeof(char) == 1`。一些常见类型的大小如右图所示。这是在 Windows 平台上,使用微软编译器时类型的宽度。对于整数和浮点数,C++ 都提供了不同宽度的类型。但除非有很好的理由,否则最好只使用 char、int 和 double 这几个标准宽度的类型。在大多数程序中(当然不是所有),其他整数和浮点数类型所带来的麻烦比带来的好处更多。



你可以将一个整数赋予一个浮点数。如果整数值超出了浮点类型的表示范围,则会丢失精度,例如:

```
cout << "sizes: " << sizeof(int) << ' ' << sizeof(float) << "\n";
int x = 2100000009; // large int
float f = x;
cout << x << ' ' << f << endl;
cout << setprecision(15) << x << ' ' << f << "\n";
```

在我们的计算机上,输出结果为:

```
Sizes: 4 4
2100000009 2.1e+009
2100000009 2100000000
```

float 类型和 int 类型占用相同大小的内存空间(4 个字节)。一个 float 值由一个“尾数” a (通常是 0 和 1 之间的一个数)和一个指数 b 组成($a * 10^b$),因此无法准确表示最大的 int 值(如果我们想把

最大 int 的准确值存入一个 float 值中, 尾数已经占用了所有空间, 指数根本没有位置存放了)。因此在上例中, f 只能保存尽量接近 2 100 000 009 的值, 对于最后一个 9 它已经无能为力了, 这就是输出结果是 2 100 000 000 的原因。

另一方面, 如果你将一个浮点数赋予一个整数, 会导致截断, 即小数部分(小数点之后的数字)被简单丢弃。例如:

```
float f = 2.8;
int x = f;
cout << x << ' ' << f << '\n';
```

x 的值会是 2, 而不是 3——这里并不是进行“四舍五入”。C++ 中 float 转换为 int 采用的是截断而非舍入。

当进行计算时, 你必须清楚可能会发生的溢出和截断。C++ 不会捕获这些问题, 考虑下面的程序:

```
void f(int i, double fpd)
{
    char c = i;           // yes: chars really are very small integers
    short s = i;          // beware: an int may not fit in a short int
    i = i+1;               // what if i was the largest int?
    long lg = i*i;         // beware: a long may not be any larger than an int
    float fps = fpd;       // beware: a large double may not fit in a float
    i = fpd;               // truncates: e.g., 5.7 -> 5
    fps = i;               // you can lose precision (for very large int values)
}

void g()
{
    char ch = 0;
    for (int i = 0; i < 500; ++i)
        cout << int(ch++) << '\t';
}
```

如果对这段程序有疑问, 尝试运行它! 对这类问题, 垂头丧气或者仅仅依赖文档都是不可取的, 实验是最好的方法!

试一试 运行 g()。修改 f(), 打印 c、s、i 等。用不同的值来测试这个函数。

我们在 25.5.3 节中还会更详细地介绍各种整数类型及它们之间的转换。如有可能, 应该使用尽可能少的类型, 这会减少混乱。例如, 如果在程序中只使用 double, 而不用 float, 就减少了可能的 double 到 float 的转换问题。实际上, 我们倾向于只使用 int、double 和 complex(参见 24.8 节)进行算术计算, 只使用 char 用于字符处理, 而 bool 用于逻辑运算, 除非迫不得已, 否则不使用其他类型。

24.2.1 数值限制

每种 C++ 的实现都在 <limits>、<limits.h> 和 <float.h> 中指明了内置类型的属性, 因此程序员可以利用这些属性来检查数值限制、设置哨兵机制等。附录 B.9.1 中列出了这些值, 它们对于开发底层程序是非常重要的。如果你觉得需要这些属性值, 表明你的工作很可能比较靠近硬件。但这些属性还有其他用途, 例如, 对语言实现细节感到好奇是很正常的: “一个 int 有多大?” 或 “char 是有符号的吗?” 希望从系统文档中找到这些问题的正确答案是很困难的, 而 C++ 标准对这类问题大多没有明确规定。较好的办法是写一个简短的小程序来获得这些问题的答案, 如下所示:

```
cout << "number of bytes in an int: " << sizeof(int) << '\n';
cout << "largest int: " << INT_MAX << endl;
cout << "smallest int value: " << numeric_limits<int>::min() << '\n';
```

```
if (numeric_limits<char>::is_signed)
    cout << "char is signed\n";
else
    cout << "char is unsigned\n";
```

```
cout << "char with min value: " << numeric_limits<char>::min() << '\n';
cout << "min char value: " << int(numeric_limits<char>::min()) << '\n';
```

如果你编写的程序将来要用在多种硬件平台上,那么能在程序中获取上面这些信息就非常有价值了。另一种方法是将这些信息硬编码到程序中,但这对维护人员来说是灾难性的。

这些属性值对溢出检测也是很有用的。

24.3 数组

数组(array)就是一个元素序列,我们可以通过下标(位置)来访问元素。我们通常也把这种数据结构称为向量。我们特别关注的一种数组是:每个元素本身也是一个数组,即多维数组,通常也称为矩阵。术语的多样性是一个概念的流行程度和使用广泛程度的标志。标准库中的 vector (参见附录 B.4)、array (参见 20.9 节)和内置数组类型(参见附录 A.8.2)都是一维的。那么,如果我们需二维数组(比如矩阵)的话,应该怎么办?如果我们需七维数组呢?

我们可以将一维和二维数组想象为如下结构:



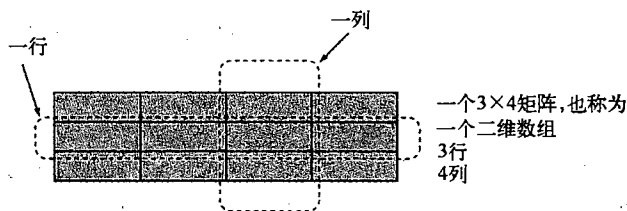
一个向量(Matrix<int>(4)),也称为
一个一维数组,或者一个 $1 \times N$ 矩阵



一个 3×4 矩阵(如Matrix<int,2>m(3,4),
也称为一个二维数组

数组对于大多数计算问题(“数值运算”)来说都是非常重要的数据结构,很多有趣的科学计算、工程计算、统计运算以及金融计算都极大地依赖于数组。

我们通常把数组看做行和列组成的结构,如下所示:



一个 3×4 矩阵,也称为
一个二维数组
3行
4列

一列就是 x 坐标相同的元素的序列,一行就是 y 坐标相同的元素的序列。

24.4 C 风格的多维数组

利用 C++ 内置的数组类型也可创建多维数组,方法是 multidimensional array 简单地看做数组的数组,即数组的元素也是数组。例如:

```

int ai[4];           // 1-dimensional array
double ad[3][4];     // 2-dimensional array
char ac[3][4][5];    // 3-dimensional array
ai[1] = 7;
ad[2][3] = 7.2;
ac[2][3][4] = 'c';

```

这种方法继承了一维数组的优点和缺点：

- 优点
 - 直接映射到硬件。
 - 低层操作效率高。
 - 语言直接支持。
- 缺点
 - C 风格的多维数组是数组的数组(见下文)。
 - 大小是固定的(即在编译时就固定下来)。如果希望在运行时再确定大小,就必须使用动态内存分配。
 - 不能干净地传递数组参数,只能转换为指向其首元素的指针。
 - 没有越界检查。通常,数组不知道它自己的大小。
 - 没有数组的整体运算,甚至没有赋值(拷贝)。

内置数组类型广泛用于数值计算,但同时也是造成程序错误和程序过于复杂的主要原因。对于大多数人来说,编写和调试使用内置数组的程序都是很痛苦的。如果你不得不使用内置数组,请查找相关资料(如《The C++ Programming Language》的附录 C, 836 ~ 840 页)。不幸的是, C++ 使用与 C 相同的内置多维数组,因此还有很多“在别处”的代码在使用这种数组。

内置数组最大的问题是不能干净地传递多维数组参数,必须转而使用指针,并显式计算数组元素位置。例如:

```

void f1(int a[3][5]);           // useful for [3][5] matrices only

void f2(int [ ][5], int dim1);   // 1st dimension can be a variable

void f3(int [5][ ], int dim2);   // error: 2nd dimension cannot be a variable

void f4(int[ ][ ], int dim1, int dim2); // error (and wouldn't work anyway)
void f5(int* m, int dim1, int dim2) // odd, but works
{
    for (int i=0; i<dim1; ++i)
        for (int j = 0; j<dim2; ++j) m[i*dim2+j] = 0;
}

```

在这段程序中,虽然 m 是一个二维数组,但我们只能将它作为 int^* 类型的参数来传递。只要数组的第二维大小是可变的(作为一个参数),就无法告知编译器参数 m 是一个 $(\text{dim1}, \text{dim2})$ 数组,而只能传递指向其起始地址的指针。表达式 $m[i * \text{dim2} + j]$ 实际就表示 $m[i, j]$,但由于编译器不知道 m 是一个二维数组,我们不得不显式计算 $m[i, j]$ 在内存中的位置。

以我们的观点来看,这太麻烦、太原始,也太容易出错了。而且运行速度也可能很慢,因为显式计算元素地址会使代码优化更为复杂。因此,我们不再介绍内置多维数组,而是重点讨论 Matrix 库中的多维数组机制,它没有上述缺点。

24.5 Matrix 库

如果以数值计算为目标的话,我们到底希望从数组/矩阵库中获得什么呢?

- “程序中使用数组的方式应该与数学/工程教科书上对数组的使用方式相近”
 - 向量、矩阵、张量等
- 具备编译时和运行时检查功能
 - 支持任意维数组
 - 支持每一维任意多个元素
- 数组是真正的变量/对象
 - 可以作为参数传递
- 支持常见的数组运算
 - 下标: ()
 - 子数组: []
 - 赋值: =
 - 标量运算 (+ =、- =、* =、% = 等)
 - 融合的向量运算 (如 `res[i] = a[i] * c + b[2]`)
 - 点积 (`res = a[i] * b[i]` 的和, 也称为内积)
- 将传统的数组/向量的概念转换为代码, 这些代码要是你自己来写的话, 会花费极大的精力, 而且效率也不会比现在的更好。
- 如果需要, 你可以扩展它 (也就是说, 库的实现没有使用什么“魔法”)。

Matrix 实现了上述功能, 也只实现了这些。如果你需要更多功能, 例如高级的数组函数、稀疏数组、控制内存布局等, 可以自己编写相应的程序或者选用一个更接近你要求的库 (第二种方式更好些)。但是, 很多这些“高级”功能可以通过在 Matrix 之上构造算法和数据结构来实现。Matrix 库不是 ISO C++ 标准库的一部分。你可以在课程网站上找到 `Matrix.h`, 整个库定义在名字空间 `Numeric_lib` 中。我们选择“矩阵”作为库的名字, 是因为“向量”和“数组”在 C++ 标准库中已经用得太多了。在英语中, 矩阵 `matrix` 的复数形式是 `matrices`, `matrixes` 也是正确的, 但很少使用。由于“Matrix”指的是一个 C++ 语言实体, 因此在本书的英文原版中使用 `Matrixes`, 以避免混淆。Matrix 库的实现使用了一些高级技术, 因此我们不会对此进行介绍。

24.5.1 矩阵的维和矩阵访问

考察下面的简单例程:

```
#include "Matrix.h"
using namespace Numeric_lib;

void f(int n1, int n2, int n3)
{
    Matrix<double,1> ad1(n1); // elements are doubles; one dimension
    Matrix<int,1> ai1(n1);    // elements are ints; one dimension
    ad1(7) = 0;              // subscript using () — Fortran style
    ad1[7] = 8;              // [] also works — C style

    Matrix<double,2> ad2(n1,n2); // 2-dimensional
    Matrix<double,3> ad3(n1,n2,n3); // 3-dimensional
    ad2(3,4) = 7.5;            // true multidimensional subscripting
    ad3(3,4,5) = 9.2;
}
```

可以看到, 你可以定义 Matrix 对象, 指定元素类型以及维数。显然, Matrix 是一个模板, 元素类型和维数是模板参数。给定 Matrix 两个模板参数 (如 `Matrix < double, 2 >`) 后, 就得到一个具体类型

(类), 你可以利用它来定义对象(如 `Matrix<double, 2> ad2(n1, n2)`), 其中的参数指定了矩阵的维。这样就定义了一个二维数组 `ad2`, 两个维度的大小分别为 `n1` 和 `n2`。我们可以使用下标操作从 `Matrix` 中获取元素, 对于一维 `Matrix`, 指定一个下标即可; 对于二维 `Matrix`, 需指定两个下标; 依此类推。

与内置数组类型和 `vector` 相似, `Matrix` 的下标是从 0 开始的(Fortran 语言从 1 开始)。也就是说, `Matrix` 元素的下标范围是 `[0, max)`, 其中 `max` 是元素总数。

这种方式很简单, 而且“完全出自于教科书”。如果你对这点有疑问, 你可以查阅适合的数学教科书, 而不是程序设计手册。这里唯一的“小聪明”是, 你可以省略维数: 默认值是一维数组。注意, 下标操作既可以使用 `[]` (C 和 C++ 风格), 也可以使用 `()` (Fortran 风格), 这使我们能更好地处理多维数组。`[x]` 下标运算符最多接受一个下标, 得到矩阵的一行; 如果 `a` 是 `n` 维矩阵, 则 `a[x]` 为 `n-1` 维矩阵。`(x, y, z)` 运算符接受一个或多个下标, 得到矩阵的一个元素, 下标的数目必须与维数相等。

下面程序给出了 `Matrix` 的一些错误用法:

```
void f(int n1, int n2, int n3)
{
    Matrix<int,0> ai0;      // error: no 0D matrices

    Matrix<double,1> ad1(5);
    Matrix<int,1> ai(5);
    Matrix<double,1> ad11(7);

    ad1(7) = 0;           // Matrix_error exception (7 is out of range)
    ad1 = ai;             // error: different element types
    ad1 = ad11;           // Matrix_error exception (different dimensions)

    Matrix<double,2> ad2(n1); // error: length of 2nd dimension missing
    ad2(3) = 7.5;           // error: wrong number of subscripts
    ad2(1,2,3) = 7.5;       // error: wrong number of subscripts

    Matrix<double,3> ad3(n1,n2,n3);
    Matrix<double,3> ad33(n1,n2,n3);
    ad3 = ad33;            // OK: same element type, same dimensions
}
```

一种错误是声明的维数与使用的维数不符, 这种错误会在编译时被捕获。而范围错误则在运行时被捕获, 程序会抛出一个 `Matrix_error` 异常。

二维矩阵的第一维是行, 第二维是列, 因此可用 (row, column) 来索引二维矩阵 (二维数组)。也可以使用 `[row][column]`, 因为对于二维矩阵, 使用单个下标会得到一个一维矩阵 (行), 如右图所示。此矩阵在内存中以“行主次序”存放:

				$a[1][2]$	
$a[0]:$	00	01	02	03	$a(1,2)$
$a[1]:$	10	11	12	13	
$a[2]:$	20	21	22	23	

00	01	02	03	10	11	12	13	20	21	22	23
----	----	----	----	----	----	----	----	----	----	----	----

一个 `Matrix` 对象是“知道”自己的维数和每维大小的, 因此, 将 `Matrix` 对象作为参数传递是很简单的:

```
void init(Matrix<int,2>& a) // initialize each element to a characteristic value
{
    for (int i=0; i<a.dim1(); ++i)
        for (int j=0; j<a.dim2(); ++j)
            a(i,j) = 10*i+j;
```

```

}

void print(const Matrix<int,2>& a) // print the elements of a row by row
{
    for (int i=0; i<a.dim1(); ++i) {
        for (int j = 0; j<a.dim2(); ++j)
            cout << a(i,j) << '\t';
        cout << '\n';
    }
}

```

可以看到, `dim1()` 返回第一维的元素数目, `dim2()` 为第二维的元素数目, 依此类推。元素类型和维数是 `Matrix` 类型的一部分, 因此函数参数不能是任意 `Matrix` (但模板参数可以):

```
void init(Matrix& a); // error: element type and number of dimensions missing
```

注意, `Matrix` 库不支持矩阵整体运算, 如将两个四维矩阵相加, 或者将一个二维矩阵和一个一维矩阵相乘。为这些运算设计优美而高效的算法超出了当前这个库的范围, 但我们可以在 `Matrix` 库之上设计这些算法 (参见习题 12)。

24.5.2 一维矩阵

我们可以对最简单的 `Matrix`——一维 `Matrix` 进行什么操作呢?

如前所述, 声明时可以省略维数:

```

Matrix<int,1> a1(8); // a1 is a 1D Matrix of ints
Matrix<int> a(8);    // means Matrix<int,1> a(8);

```

即 `a` 和 `a1` 是相同的类型 (`Matrix<int, 1>`)。我们可以获取矩阵的大小 (元素总数) 和每一维的大小 (一维中的元素数目), 对于一维矩阵, 这两个值显然是相同的:

```

a.size(); // number of elements in Matrix
a.dim1(); // number of elements in 1st dimension

```

我们可以按内存中的实际布局获取元素, 即获得指向第一个元素的指针:

```
int* p = a.data(); // extract data as a pointer to an array
```

如果希望将 `Matrix` 对象传递给只接受指针参数的 C 风格的函数, 这个操作是很有用的。我们可以像下面代码这样对矩阵进行下标操作:

```

a(i); // ith element (Fortran style), but range checked
a[i]; // ith element (C style), range checked
a(1,2); // error: a is a 1D Matrix

```

一些算法常常需要访问子矩阵, `slice()` 就完成这一功能, 它有两种形式:

```

a.slice(i); // the elements from a[i] to the last
a.slice(i,n); // the n elements from a[i] to a[i+n-1]

```

下标和子矩阵操作既可以作为右值, 也可以作为左值, 因为它们直接指向矩阵的元素, 而不是创建拷贝。例如:

```
a.slice(4,4) = a.slice(0,4); // assign first half of a to second half
```

如果 `a` 的初值为

```
{ 1 2 3 4 5 6 7 8 }
```

则执行这条语句后, `a` 变为:

```
{ 1 2 3 4 1 2 3 4 }
```

注意, 最常用的子矩阵是“开始元素段”和“末尾元素段”, 即 `a.slice(0, j)` (区域 `[0:j)`) 和 `a.slice(j)` (区域 `[j:a.size())`)。特别地, 上面那条语句可以写为:

```
a.slice(4) = a.slice(0,4); // assign first half of a to second half
```

也就是说,语法的设计上更倾向于常用情况。你可以指定超出 a 的范围的 i 和 n 的值,但最终的结果只取 a 的有效范围内的那段。例如, $a.slice(i, a.size())$ 实际得到范围是 $[i:a.size())$, 而 $a.slice(a.size())$ 和 $a.slice(a.size(), 2)$ 得到的则是空矩阵。对于很多算法来说,这一特性在某些时候是很有用的。这个特性实际上是从数学领域借鉴来的。显然, $a.slice(i, 0)$ 是空的,我们不会故意写出这样的代码,但算法中使用 $a.slice(i, n)$ 而 n 恰巧为 0 的情况是很可能出现的。如果此时能得到空矩阵而不是产生一个错误的话,算法可以更为简洁。

Matrix 也支持(对 C++ 对象来说)常见的拷贝操作,实现所有元素的复制:

```
Matrix<int> a2 = a;    // copy initialization
a = a2;               // copy assignment
```

我们可以对矩阵中每个元素进行相同的内置运算(标量运算):

```
a *= 7;               // scaling: a[i]*=7 for each i (also +=, -=, /=, etc.)
a = 7;               // a[i]=7 for each i
```

只要元素类型支持,其他赋值运算符和组合赋值运算符($=$ 、 $+=$ 、 $-=$ 、 $/=$ 、 $*=$ 、 $\% =$ 、 $^ =$ 、 $\& =$ 、 $| =$ 、 $>> =$ 、 $<< =$)也都可以这样使用。我们还可以对矩阵每个元素执行相同的函数:

```
a.apply(f);          // a[i]=f(a[i]) for each element a[i]
a.apply(f,7);        // a[i]=f(a[i],7) for each element a[i]
```

组合赋值运算符和函数 `apply()` 都修改了 Matrix 中的元素,如果你不希望这样,而是希望创建一个新的 Matrix 对象保存运算结果,可以这样做:

```
b = apply(abs,a);    // make a new Matrix with b(i)==abs(a(i))
```

其中的 `abs` 是标准库中的绝对值函数(参见 24.8 节)。本质上, `apply(f, x)` 与 `x.apply(f)` 相对应,而 $+$ 与 $+=$ 对应。例如:

```
b = a*7;             // b[i] = a[i]*7 for each i
a *= 7;              // a[i] = a[i]*7 for each i
y = apply(f,x);      // y[i] = f(x[i]) for each i
x.apply(f);          // x[i] = f(x[i]) for each i
```

其运行结果为 $a == b$ 且 $x == y$ 。

在 Fortran 中,第二个版本的 `apply` 称为“广播”函数,通常写作 $f(x)$ 而不是 `apply(f, x)`。为了使这一特性对任意函数 f 都适用(而不是像 Fortran 中那样,只对少数函数适用),我们需要为“广播”操作定义一个名字,因此(再次)使用了 `apply`。

另外,接受两个参数的版本如下:

```
b = apply(f,a,x);    // b[i]=f(a[i],x) for each i
```

使用方式如下所示:

```
double scale(double d, double s) { return d*s; }
b = apply(scale,a,7); // b[i] = a[i]*7 for each i
```

注意,“独立式”`apply()` 接受一个函数作为参数,并通过其参数生成运算结果,然后利用运算结果初始化结果矩阵。它通常不修改参数中的 Matrix 对象。成员函数 `apply` 的不同之处在于,会修改原矩阵中元素。例如:

```
void scale_in_place(double& d, double s) { d *= s; }
b.apply(scale_in_place,7); // b[i] *= 7 for each i
```

Matrix 库还支持传统数值计算库中一些最常用的函数:

```
Matrix<int> a3 = scale_and_add(a,8,a2); // fused multiply and add
int r = dot_product(a3,a);             // dot product
```

函数 `scale_and_add()` 通常称为融合乘-加运算(fused multiply-add, fma),它对矩阵中每个元素 i 执行 $result(i) = arg1(i) * arg2 + arg3(i)$ 。点积运算 `dot_product` 也称为内积 `inner_product`, 我们已

经在 21.5.3 节中对这种运算进行了介绍,它对矩阵中每个元素执行 $\text{result} += \text{arg1}(i) * \text{arg2}(i)$, result 的初值为 0。

一维数组是非常常用的,可以用内置数组类型、vector 或者 Matrix 来实现。我们之所以使用 Matrix 库,更多的是因为需要进行矩阵的整体运算,如 $*$, 或者需要使用多维矩阵。

像 Matrix 库函数这类工具,可以描述为“与数学描述非常接近”或者“令程序员不必编写循环来处理矩阵中的每个元素”。总之,利用这些函数编写程序,代码会非常简洁,而且不容易出错。Matrix 库提供的那些操作,如拷贝、为所有元素赋值以及对所有元素执行相同运算等,都使我们不必编写、维护循环代码(也不必为写出的循环是否确实正确而烦恼)。

Matrix 提供了两个构造函数,将数据从内置数组复制到 Matrix 对象中,如下所示:

```
void some_function(double* p, int n)
{
    double val[] = { 1.2, 2.3, 3.4, 4.5 };
    Matrix<double> data(p,n);
    Matrix<double> constants(val);
    // ...
}
```

如果数据是来自于某个未使用 Matrix 库的程序片段,是以数组或 vector 的形式提供的,这两个构造函数就非常有用。

注意,编译器能够推断出已经初始化的数组的规模,因此上面这段程序在定义 constants 时无须给出元素个数。另一方面,如果只是给出一个指针的话,编译器是无法获得元素数目的,因此定义 data 时,必须给出指针 p 指向的数组的规模(n)。

24.5.3 二维矩阵

Matrix 库的设计思想是:不同维数的矩阵除了维数之外,其他方面实际上是非常相似的,因此,很多一维数组的概念都可用于二维数组,如下所示:

```
Matrix<int,2> a(3,4);

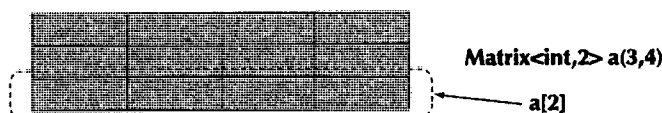
int s = a.size();      // number of elements
int d1 = a.dim1();     // number of elements in a row
int d2 = a.dim2();     // number of elements in a column
int* p = a.data();     // extract data as a pointer to a C-style array
```

可以看到,我们能够获取元素总数和每一维的元素数目,可以获取矩阵在内存中存储区域的指针。

当然,下标操作仍然是必不可少的,如下所示:

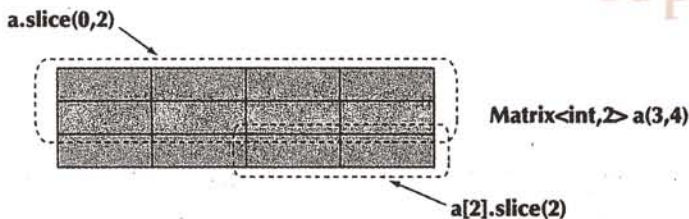
```
a(i,j);      // (i,j)th element (Fortran style), but range checked
a[i];        // ith row (C style), range checked
a[i][j];     // (i,j)th element (C style)
```

对于一个二维矩阵,下标操作 $[i]$ 获得它的第 i 行,即一个一维矩阵。这意味着,我们可以利用这一特性从二维矩阵中提取行,传递给那些需要一维矩阵甚至内置数组 ($a[i].data()$) 参数的操作或者函数。注意, $a(i, j)$ 可能比 $a[i][j]$ 更快,虽然这完全由编译器和优化器所决定。



二维矩阵也支持子矩阵操作,例如:

```
a.slice(i);           // the rows from the a[i] to the last
a.slice(i,n);         // the rows from the a[i] to the a[i+n-1]
```



注意,一个二维矩阵的子矩阵仍是二维矩阵(可能行数更少)。

二维矩阵的标量操作与一维矩阵类似,这些操作并不关心元素是如何组织的,只是简单地按元素在内存中存放的次序对它们进行处理而已,如下所示

```
Matrix<int,2> a2 = a; // copy initialization
a = a2;              // copy assignment
a *= 7;              // scaling (and +=, -=, /=, etc.)
a.apply(f);          // a(i,j)=f(a(i,j)) for each element a(i,j)
a.apply(f,7);        // a(i,j)=f(a(i,j),7) for each element a(i,j)
b=apply(f,a);        // make a new Matrix with b(i,j)=f(a(i,j))
b=apply(f,a,7);      // make a new Matrix with b(i,j)=f(a(i,j),7)
```

行交换常常是很有用的, Matrix 库也支持这种操作:

```
a.swap_rows(1,2); // swap rows a[1] <-> a[2]
```

Matrix 并没有提供 swap_columns() 操作,你可以自己实现它(参见习题 11)。原因在于元素是按行主次序存储的,行和列并不是完全对称的。这种不对称性也体现在[i]得到矩阵的一行,但 Matrix 并未提供提取列的运算符。在下标操作(i, j, k)中,第一个下标 i 对应第 i 行。这种不对称性也反映了深层次的数学性质。

在现实世界中,有很多事物都是二维结构的,因此显然可以用二维矩阵来描述:

```
enum Piece { none, pawn, knight, queen, king, bishop, rook };
Matrix<Piece,2> board(8,8); // a chessboard
```

```
const int white_start_row = 0;
const int black_start_row = 7;
```

```
Piece init_pos[] = {rook, knight, bishop, queen, king, bishop, knight, rook};
Matrix<Piece> start_row(init_pos); // initialize elements from init_pos
Matrix<Piece> clear_row(8); // 8 elements of the default value
```

对 clear_row 的初始化利用了两个事实: none == 0; 元素默认情况下被初始化为 0。对于 start_row 的初始化,我们可能希望写成这样:

```
Matrix<Piece> start_row
= {rook, knight, bishop, queen, king, bishop, knight, rook};
```

但是,这种语法直到下一个 C++ 标准被批准前都是不合法的,因此我们必须先初始化一个数组(本例中是 init_pos),然后再用它来初始矩阵对象。有了上述定义后,我们就可以使用 start_row 和 clear_row 了:

```
board[white_start_row] = start_row; // reset white pieces
for (int i = 1; i < 7; ++i) board[i] = clear_row; // clear middle of the board
board[black_start_row] = start_row; // reset black pieces
```

注意,当我们使用[i]提取一行时,我们得到一个左值(参见 4.3 节),即我们可以对其赋值。

24.5.4 矩阵 I/O

Matrix 库为一维和二维矩阵提供了非常简单的 I/O 功能:

```
Matrix<double> a(4);
cin >> a;
cout << a;
```

这段代码会读取以空白符间隔的 4 个 double 值, 以花括号起止, 例如:

```
{1.2 3.4 5.6 7.8}
```

输出结果与输入内容很相似, 因此你所写入的数据可以按相同的方式读取出来。

二维矩阵的 I/O 操作简单地读写花括号包含起来的一维矩阵序列, 例如:

```
Matrix<int,2> m(2,2);
cin >> m;
cout << m;
```

这段代码读取类似下面所示的内容:

```
{
{1 2}
{3 4}
}
```

输出操作与输入操作非常接近。

矩阵的 << 和 >> 操作符主要是为了方便编写简单程序。如果你有更高的要求, 就只能自己实现了。另外, << 和 >> 的定义是在 MatrixIO.h 中(而不是 Matrix.h), 因此, 只是使用矩阵的基本功能(而不使用 I/O 功能)的话, 就不需要包含此头文件了。

24.5.5 三维矩阵

三维(以及更高维)矩阵与二维矩阵相比, 除了维数更多外, 其他方面非常相似。看看下面的代码:

```
Matrix<int,3> a(10,20,30);

a.size();           // number of elements
a.dim1();           // number of elements in dimension 1
a.dim2();           // number of elements in dimension 2
a.dim3();           // number of elements in dimension 3
int* p = a.data();  // extract data as a pointer to a C-style array
a(i,j,k);           // (i,j,k)th element (Fortran style), but range checked
a[i];               // ith row (C style), range checked
a[i][j][k];         // (i,j,k)th element (C style)
a.slice(i);          // the elements from the ith to the last
a.slice(i,j);        // the elements from the ith to the jth
Matrix<int,3> a2 = a; // copy initialization
a = a2;              // copy assignment
a *= 7;              // scaling (and +=, -=, /=, etc.)
a.apply(f);          // a(i,j,k)=f(a(i,j,k)) for each element a(i,j,k)
a.apply(f,7);        // a(i,j,k)=f(a(i,j,k),7) for each element a(i,j,k)
b=apply(f,a);         // make a new Matrix with b(i,j,k)=f(a(i,j,k))
b=apply(f,a,7);       // make a new Matrix with b(i,j,k)=f(a(i,j,k),7)
a.swap_rows(7,9);    // swap rows a[7] <-> a[9]
```

如果你理解二维矩阵的相关概念, 那么也就能理解三维矩阵了。例如, 在这段程序中, a 是一个三维矩阵, 那么 a[i] 就是一个二维矩阵(如果 i 是合法下标), a[i][j] 就是一个一维矩阵(如果 j 是合法下标), 而 a[i][j][k] 就是一个整型元素(如果 k 是合法下标)。

我们倾向于把现实世界看成三维的, 因此三维矩阵显然可以用于现实世界的建模(例如, 使

用笛卡尔坐标系进行物理仿真):

```
int grid_nx;    // grid resolution; set at startup
int grid_ny;
int grid_nz;
Matrix<double,3> cube(grid_nx, grid_ny, grid_nz);
```

如果我们将时间加入, 作为第四维, 那么就得到了一个四维空间, 可用一个四维 Matrix 描述, 依此类推。

24.6 实例: 求解线性方程组

对于一个数值计算程序, 如果你能理解代码背后的数学含义, 它对你来说就是有意义的, 否则, 它就是废话一堆。本节给出的例子是求解线性方程组问题, 如果你学习过基本的线性代数知识, 会觉得它很简单; 否则, 你就把它看做求解方案到代码的简单转换就可以了。

求解线性方程组是矩阵的一个相当实际也非常重要的应用, 其目标是求解下面这种形式的线性方程组:

$$\begin{aligned} a_{1,1}x_1 + \cdots + a_{1,n}x_n &= b_1 \\ &\vdots \\ a_{n,1}x_1 + \cdots + a_{n,n}x_n &= b_n \end{aligned}$$

其中 x_1, \dots, x_n 表示 n 个未知数, $a_{1,1}, \dots, a_{n,n}$ 和 b_1, \dots, b_n 是给定的常量。简单起见, 我们假定未知数和常量都是浮点值。问题的目标是找到能同时满足 n 个方程的未知数值。方程组可以更简洁地表示为矩阵和向量乘法的形式:

$$Ax = b$$

其中, A 是一个 $n \times n$ 的系数方阵:

$$A = \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{bmatrix}$$

向量 x 和 b 分别是未知数和常量向量:

$$x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

这个系统可能有 0 个、1 个或者无穷多个解, 这取决于系数矩阵 A 和向量 b 。求解线性系统的方法有很多, 本节使用一种经典的方法——高斯消去法(参见 Freeman 和 Phillips 的《Parallel Numerical Algorithms》; Stewart 的《Matrix Algorithms(第一卷)》以及 Wood 的《Introduction to Numerical Analysis》)。首先, 我们对 A 和 b 进行变换, 使得 A 变为一个上三角矩阵。所谓上三角矩阵, 就是对角线之下的所有元素均为 0。换句话说, 系统转换为如下形式:

$$\begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ 0 & \ddots & \vdots \\ 0 & 0 & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

实现这个目标是很容易的。为了使 $a(i, j)$ 变为 0, 我们先将它乘以一个常量, 使它等于第 j 列上的另一个元素, 比如说等于 $a(k, j)$ 。然后, 用第 i 个方程减去第 k 个方程, $a(i, j)$ 即变为 0, 矩阵第 i 行其他元素的值也相应发生改变。

如果这样一个变换最终使得对角线上所有元素都非 0, 方程组就有唯一解, 此解可以通过“回代”(back substitution)求得。其过程是这样的, 首先通过最后一个方程求得 x_n :

$$a_{n,n}x_n = b_n$$

显然, $x_n = b_n / a_{n,n}$ 。随后, 将第 n 行从系统中删除, 继续求解 x_{n-1} , 依次类推, 直至求解出 x_1 的值。在每个步骤中, 都是除以 $a_{i,i}$, 因此对角线上的元素必须非 0。否则, 回代方法就失败了, 意味着方程组有 0 个或无穷多个解。

24.6.1 经典的高斯消去法

我们现在来看看如何用 C++ 程序来表示上述计算方法。首先, 定义两个要使用的具体 Matrix 类型, 以简化程序:

```
typedef Numeric_lib::Matrix<double, 2> Matrix;
typedef Numeric_lib::Matrix<double, 1> Vector;
```

接下来我们将高斯消去法计算过程描述如下:

```
Vector classical_gaussian_elimination(Matrix A, Vector b)
{
    classical_elimination(A, b);
    return back_substitution(A, b);
}
```

即先为两个输入 A 和 b 创建拷贝(使用传值参数), 然后调用一个函数求解方程组, 最后调用回代函数计算结果并将结果返回。关键之处在于, 我们分解问题的方式和符号表示都完全来自于原始的数学描述。下面所要做的就是实现 `classical_elimination()` 和 `back_substitution()` 了, 解决方案同样完全来自于数学教科书:

```
void classical_elimination(Matrix& A, Vector& b)
{
    const Index n = A.dim1();

    // traverse from 1st column to the next-to-last
    // filling zeros into all elements under the diagonal:
    for (Index j = 0; j < n-1; ++j) {
        const double pivot = A(j, j);
        if (pivot == 0) throw Elim_failure(j);

        // fill zeros into each element under the diagonal of the ith row:
        for (Index i = j+1; i < n; ++i) {
            const double mult = A(i, j) / pivot;
            A[i].slice(j) = scale_and_add(A[j].slice(j), -mult, A[i].slice(j));
            b(i) -= mult * b(j); // make the corresponding change to b
        }
    }
}
```

“pivot”表示当前行位于对角线上的元素, 它必须非 0, 因为需要用它作为除数; 如果它为 0, 我们将放弃计算, 抛出一个异常:

```
Vector back_substitution(const Matrix& A, const Vector& b)
{
    const Index n = A.dim1();
    Vector x(n);

    for (Index i = n-1; i >= 0; --i) {
        double s = b(i) - dot_product(A[i].slice(i+1), x.slice(i+1));

        if (double m = A(i, i))
            x(i) = s / m;
        else
            throw Back_subst_failure(i);
    }

    return x;
}
```

24.6.2 选取主元

pivot 为 0 的问题是可以避免的, 我们可以对行进行排序, 从而将 0 和较小的值从对角线上移开, 这样就得到了一个更鲁棒的方案。“更鲁棒”是指对于舍入误差更不敏感。但是, 随着我们将 0 置于对角线之下, 元素值也会发生改变。因此, 为了降低误差的影响, 我们必须进行重排序, 以将较小的值从对角线上移开(即不能重排矩阵后就直接使用经典算法):

```
void elim_with_partial_pivot(Matrix& A, Vector& b)
{
    const Index n = A.dim1();

    for (Index j = 0; j < n; ++j) {
        Index pivot_row = j;

        // look for a suitable pivot:
        for (Index k = j + 1; k < n; ++k)
            if (abs(A(k, j)) > abs(A(pivot_row, j))) pivot_row = k;

        // swap the rows if we found a better pivot:
        if (pivot_row != j) {
            A.swap_rows(j, pivot_row);
            std::swap(b(j), b(pivot_row));
        }

        // elimination:
        for (Index i = j + 1; i < n; ++i) {
            const double pivot = A(j, j);
            if (pivot == 0) error("can't solve: pivot==0");
            const double mult = A(i, j)/pivot;
            A[i].slice(j) = scale_and_add(A[j].slice(j), -mult, A[i].slice(j));
            b(i) -= mult * b(j);
        }
    }
}
```

在这里我们使用了 `swap_rows()` 和 `scale_and_multiply()`, 这样程序更符合习惯, 我们也不必显式编写循环代码了。

24.6.3 测试

显然, 我们下面应该对代码进行测试。幸运的是, 我们有一种简单的测试方法, 如下所示:

```
void solve_random_system(Index n)
{
    Matrix A = random_matrix(n); // see §24.7
    Vector b = random_vector(n);

    cout << "A = " << A << endl;
    cout << "b = " << b << endl;

    try {
        Vector x = classical_gaussian_elimination(A, b);
        cout << "classical elim solution is x = " << x << endl;
        Vector v = A * x;
        cout << "A * x = " << v << endl;
    }
    catch(const exception& e) {
        cerr << e.what() << std::endl;
    }
}
```

程序在三种情况下会进入 catch 子句：

- 代码中有 bug(但是,作为乐观主义者,我们不认为现在的代码中有 bug)。
- 输入内容是 classical_elimination 出现错误(我们本应该用 elim_with_partial_pivot)。
- 舍入误差导致问题。

但是,这种测试方法不像我们期望的那样接近实际,因为真正的随机矩阵不太可能导致 classical_elimination 出现错误。

为了测试程序,我们输出 $A * x$, 其值应该与 b 相当。但考虑到存在舍入误差,若其值与 b 足够接近就应该认为结果正确,这也是为什么测试程序中没有采用下面语句来判断结果是否正确的原因:

```
if (A*x!=b) error("substitution failed");
```

在计算机中,浮点数只是实数的近似,因此我们必须接受近似的计算结果。一般来说,应该避免使用 `==` 和 `!=` 来判断结果是否正确。

Matrix 库并没有定义矩阵与向量的乘法运算,因此我们为测试程序定义这个运算:

```
Vector operator*(const Matrix& m, const Vector& u)
{
    const Index n = m.dim1();
    Vector v(n);
    for (Index i = 0; i < n; ++i) v(i) = dot_product(m[i], u);
    return v;
}
```

我们再次看到,一个简单 Matrix 操作能帮助我们完成大部分工作。Matrix 的输出操作是在 MatrixIO.h 中定义的,我们在 24.5.3 节中已经对此进行了介绍。random_matrix() 和 random_vector() 是随机数的简单应用(参见 24.7 节),这两个函数的实现留作练习。Index 是索引类型,它是用 typedef 定义的(参见附录 A.15)。我们使用 using 将 Index 引入当前作用域:

```
using Numeric_lib::Index;
```

24.7 随机数

如果你要求人们说出一个随机数,大多数人会回答 7 或 17,这表明人们认为这两个数是“最随机的”。几乎不会有人回答 0,因为 0 被视为一个非常完美的数,没人认为它是“随机的”,因此被看做“最不随机”的数。从数学的观点来看,这绝对是错误的。随机数并不是指单个的数。我们常用的、常说的随机数,是指一个服从某种分布的序列,其特点是你无法很容易地从序列前一部分的内容预测出下一个数是什么。随机数的应用领域非常广,包括程序测试(用于生成大量测试用例)、游戏(确保游戏的下一步与之前的步骤不同)以及仿真(令仿真对象在参数限定范围内“随机地”运行)等。

随机数既是一个实用工具,也是一个数学问题,它高度复杂,这与它在现实世界中的重要性是相匹配的。在本节中,我们只讨论随机数的最基本的内容,这些内容可用于简单的测试和仿真。在 `<cstdlib>` 中,标准库提供了如下特性:

```
int rand();           // returns values in the range [0:RAND_MAX]
RAND_MAX             // the largest value that rand() can produce
void srand(unsigned int); // seed the random number generator
```

反复调用 rand() 就可以生成一个整型伪随机数序列,此序列服从 $[0 : \text{RAND_MAX}]$ 内的均匀分布。我们称这个序列是“伪随机的”,因为它是由一个数学公式计算出来的,因此经过一段间隔后就会重复(即它是可预测的,并非完全随机的)。特别是,如果我们在一个程序中反复调用 rand(),那么每次执行这个程序都会得到相同的伪随机数序列。这对于程序调试是非常有用的特性。如果我们希望每次执行程序都得到不同的序列,可以在程序开头用不同的参数调用 srand()。调用

srand() 的参数不同, rand() 生成的随机数序列就不同。

例如, 考虑 24.6.3 节中用到的 random_vector()。调用 random_vector(n) 就会生成一个 Matrix <double, 1> 类型的矩阵对象, 它包含 n 个元素, 元素值都是 [0 : n] 之间的随机数:

```
Vector random_vector(Index n)
{
    Vector v(n);

    for (Index i = 0; i < n; ++i)
        v(i) = 1.0 * n * rand() / RAND_MAX;

    return v;
}
```

注意, 使用 1.0 是为了保证运算都是浮点运算。如果除以 RAND_MAX 的运算是整数除法, 那么得到的结果永远为 0, 这显然不是我们所期望的。

得到指定区间, 如 [0 : max) 内的一个整数, 是很困难的。很多人开始时都会尝试这样做:

```
int val = rand() % max;
```

在以往, 这不是一个好方法, 因为这个计算实际上是取随机数低位的值, 而很多传统的随机数发生器所生成的随机数, 在低位上并不是完全随机的。对于当前的大多数随机数发生器, 这个问题并不那么严重了, 但考虑到代码的可移植性, 最好将随机数的计算“隐藏”在一个函数中, 例如:

```
int rand_int(int max) { return rand() % max; }
```

```
int rand_int(int min, int max) { return rand_int(max - min) + min; }
```

采用这种方式, 当你发现系统中 rand() 的实现很差, 使得 rand_int() 的效果不佳时, 可以按需要修改 rand_int() 的实现, 而不必修改程序的其他部分。如果你是在开发一个工业品质的软件, 或者需要伪随机数序列服从非均匀分布时, 你可以选择更高质量的随机数库, 这类库有很多, 比如 Boost::random 就是其中之一。请完成习题 10, 你会对随机数发生器的质量有所体会。

24.8 标准数学函数

标准库中也提供了常用的标准数学函数(cos、sin、log 等), 这些函数的定义都在 <cmath> 中:

标准数学函数

abs(x)	绝对值
ceil(x)	向上取整—— $\geq x$ 的最小整数
floor(x)	向下取整—— $\leq x$ 的最大整数
sqrt(x)	平方根, x 必须是非负数
cos(x)	余弦
sin(x)	正弦
tan(x)	正切
acos(x)	反余弦, 结果取为非负数
asin(x)	反正弦, 结果取最接近 0 的值
atan(x)	反正切
sinh(x)	双曲正弦
cosh(x)	双曲余弦
tanh(x)	双曲正切
exp(x)	e 的指数
log(x)	自然对数, 即以 e 为底, x 必须是正数
log10(x)	以 10 为底的对数

标准数学函数的参数可以是如下类型: float、double、long double 和 complex(参见 24.9 节)。如果你需要做浮点计算,会发现这些函数非常有用。如果你需要了解更多细节,相关文档非常多,标准库的联机文档就可以作为一个很好的入门材料。

如果一个标准数学函数无法计算出有效结果,它会设置变量 `errno`。例如:

```
errno = 0;
double s2 = sqrt(-1);
if (errno) cerr << "something went wrong with something somewhere";
if (errno == EDOM) // domain error
    cerr << "sqrt() not defined for negative argument";
pow(very_large,2); // not a good idea
if (errno==ERANGE) // range error
    cerr << "pow(" << very_large << ",2) too large for a double";
```

如果你要做一些重要的计算,在计算完成之后应该检查 `errno` 的值,确保它仍为 0。如果 `errno` 变为非 0,一定是出现错误了。请查阅手册或者联机文档,检查哪些数学函数可以设置 `errno`,以及 `errno` 的不同的值分别代表什么错误类型。

如上例所示,`errno` 非 0 仅仅表示“什么地方发生错误了”,获得具体错误类型和错误位置还要正确检测 `errno` 的值。标准库之外的函数在发生错误时也可能设置 `errno` 的值,因此在检查 `errno` 值的时候一定要小心,以确保找到正确的错误位置。正确的方法是在调用标准库函数前确认 `errno == 0`,在函数返回后立刻检查 `errno` 的值,这样就可以保证 `errno` 的值反映了函数的错误类型。就像上例中那样,我们可以检测 `errno` 是否等于 `EDOM` 和 `ERANGE` 来判断错误类型。其中 `EDOM` 表示定义域错误(即参数错误),而 `ERANGE` 表示值域错误(即计算结果错误)。

基于 `errno` 的错误处理技术已经有很长历史了,它的产生可以追溯到第一个 C 语言数学函数出现的年代(1975 年)。

24.9 复数

复数在科学和工程计算中广泛使用。我们假定你已经了解了相关的数学知识,因此本节只介绍如何用 ISO C++ 标准库来表达复数运算。复数及其标准数学函数的定义都在 `<complex>` 中:

```
template<class Scalar> class complex {
    // a complex is a pair of scalar values, basically a coordinate pair
    Scalar re, im;
public:
    complex(const Scalar & r, const Scalar & i) :re(r), im(i) {}
    complex(const Scalar & r) :re(r),im(Scalar()) {}
    complex() :re(Scalar()), im(Scalar()) {}

    Scalar real() { return re; } // real part
    Scalar imag() { return im; } // imaginary part

    // operators: = += -= *= /=
};
```

标准库中的 `complex` 支持标量类型 float、double 和 long double 构成的复数。除了 `complex` 的成员以及标准数学函数(参见 24.8 节)之外,`<complex>` 还提供了大量有用的运算:

复数运算

$z1 + z2$	加法
$z1 - z2$	减法
$z1 * z2$	乘法

(续)

复数运算

$z1/z2$	除法
$z1 == z2$	相等
$z1 != z2$	不等
$\text{norm}(z)$	$\text{abs}(z)$ 的平方
$\text{conj}(z)$	共轭: 如果 z 是 $ re, im $, 则 $\text{conj}(z)$ 为 $ re, -im $
$\text{polar}(x, y)$	用给定的极坐标 (ρ, θ) 构造一个复数
$\text{real}(z)$	实部
$\text{imag}(z)$	虚部
$\text{abs}(z)$	模, 也称为 ρ
$\text{arg}(z)$	辐角, 也称为 θ
$\text{out} << z$	输出
$\text{in} >> z$	输入

注意, `complex` 未提供 `<` 或 `%`。

`complex <T>` 的使用与 `double` 这样的内置类型完全一样。例如:

```
typedef complex<double> dcmplx; // sometimes complex<double>
                                // gets verbose

void f(dcmplx z, vector<dcmplx>& vc)
{
    dcmplx z2 = pow(z,2);
    dcmplx z3 = z2*9.3+vc[3];
    dcmplx sum = accumulate(vc.begin(), vc.end(), dcmplx());
    // ...
}
```

请记住, 并不是对所有的 `int` 和 `double` 运算, `complex` 都定义了相应的复数运算。例如:

```
if (z2<z3) // error: there is no < for complex numbers
```

注意, C++ 标准库中复数的描述 (布局) 与 C 和 Fortran 中的对应类型是兼容的。

24.10 参考文献

实际上, 本章所讨论的话题, 如舍入误差、矩阵运算以及复数运算等, 如果孤立地看, 没有任何意义。本章只是面向那些具备一定数学基础, 需要进行数值计算的人, 简单地介绍 C++ 中的一些相关特性。

假若你在这些领域上已经有些荒废了, 或者仅仅是有些好奇, 我们向你推荐下面的资源: MacTutor 数学史档案, <http://www-gap.dcs.st-and.ac.uk/~history>

- 对所有喜欢数学或者仅仅是需要使用数学的人都是一个极好的网站。
- 对一些希望了解数学的人性化一面的人 (例如, 想知道谁是唯一一个获得过奥运会金牌的知名数学家), 这是一个极好的网站。在网站上可以找到:
 - 著名数学家的传记和成就
 - 一些“古董”

- 著名的函数曲线。
- 著名数学问题。
- 数学领域的各种主题。
 - 代数
 - 分析
 - 数论
 - 几何和拓扑学
 - 数学物理学
 - 数理天文学
 - ◆ 数学历史

◆

Freeman, T. L., and Chris Phillips. *Parallel Numerical Algorithms*. Prentice Hall, 1992.

Gullberg, Jan. *Mathematics - From the Birth of Numbers*. W. W. Norton, 1996. ISBN 039304002X. One of the most enjoyable books on basic and useful mathematics. A (rare) math book that you can read for pleasure and also use to look up specific topics, such as matrices.

Knuth, Donald E. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Third Edition*. Addison-Wesley, 1998. ISBN: 0201896842.

Stewart, G. W. *Matrix Algorithms, Volume I: Basic Decompositions*. SIAM, 1998. ISBN 0898714141.

Wood, Alistair. *Introduction to Numerical Analysis*. Addison-Wesley, 1999. ISBN 020194291X.

简单练习

1. 打印 char、short、int、long、float、double、int* 和 double* 的宽度(利用 sizeof, 而不是 <limits>)。
2. 用 sizeof 打印 Matrix <int> a(10)、Matrix <int> b(100)、Matrix <double> c(10)、Matrix <int, 2> d(10, 10)、Matrix <int, 3> e(10, 10, 10) 的宽度。
3. 打印第 2 题中每个矩阵的元素数目。
4. 编写程序, 从 cin 读入 int 型整数, 输出每个整数的平方根 sqrt(), 或者“no square root”(检查 sqrt() 的返回值, 判断运算是否非法)。
5. 从输入读取 10 个浮点值, 将它们存入一个 Matrix <double> 对象。Matrix 没有定义 push_back() 操作, 所以要小心处理错误的 double 值的情况。最后输出矩阵。
6. 计算 $[0, n) * [0, m)$ 乘法表, 将它保存在一个二维矩阵中。n 和 m 的值从 cin 读入。最后, 以漂亮的格式输出乘法表(假定 m 足够小, 屏幕的一行能容纳乘法表的一行)。
7. 从 cin 读入 10 个复数 complex <double> (cin 支持 complex 的 >> 操作), 将它们保存在一个矩阵中。计算并输出这 10 个复数的和。
8. 读入 6 个 int 型整数, 存入一个矩阵对象 Matrix <int, 2> m(2, 3) 中, 并输出这些整数。

思考题

1. 哪些人使用数值计算?
2. 什么是精度?
3. 什么是溢出?
4. 一般来说 double 的宽度是多少? int 的宽度是多少?
5. 你如何检测溢出?
6. 在哪里能找到数值限制? 如最大的 int 值是多大?
7. 什么是数组? 什么是行, 什么是列?
8. 什么是 C 风格的多维数组?
9. 程序设计语言中支持矩阵运算的部分(如矩阵库)必备的特性是什么?
10. 什么是矩阵的维?

11. 一个矩阵可以有多少维(从理论上、数学上看)?
12. 什么是子矩阵?
13. 什么是“广播”运算? 请举出一些例子。
14. Fortran 风格的下标和 C 风格的下标有什么区别?
15. 如何对矩阵中每个元素都进行一个相同的操作? 请举例。
16. 什么是融合运算?
17. 请定义点积运算。
18. 什么是线性代数?
19. 什么高斯消去法?
20. (线性代数中的、“现实生活中”的)主元(pivot)是指什么?
21. 什么令一个数随机?
22. 什么是均匀分布?
23. 从哪里可以找到标准数学函数? 它们支持哪些类型的参数?
24. 复数的虚部是什么?
25. -1 的平方根是什么?



术语

数组	Fortran	标量的	C	融合运算	大小
列	虚部	sizeof	复数	Matrix	子矩阵
维	多维	下标	点积	随机数	均匀分布
逐元素运算	实数	errno	行		



习题

1. `a.apply(f)` 和 `apply(f, a)` 所使用的函数参数 `f` 是不同的。为两个 `apply()` 分别编写一个 `double()` 函数, 完成的功能都是将数组 `{ 1 2 3 4 5 }` 中元素值加倍。定义一个统一的 `double()` 函数, 既能用于 `a.apply(double)`, 又能用于 `apply(double, a)`。解释一下, 为什么采取这种方式编写 `apply()` 所使用的函数并不是一种好的方法?
2. 重做习题 1, 但实现的是函数对象而不是函数。提示: `Matrix.h` 中有示例。
3. 本题不适合初学者(利用本书中介绍的工具无法完成此题): 编写一个 `apply(f, a)`, 对于函数参数 `f`, `apply` 可以接受的类型有 `void (T&)`、`T (const T&)` 以及对应的函数对象。提示: 参考 `Boost::bind`。
4. 编译、测试高斯消去法程序。
5. 用 `A == { {0 1} {1 0} }` 和 `b == {5 6}` 测试高斯消去法程序, 观察错误。然后测试 `elim_with_partial_pivot()`。
6. 将高斯消去法程序中的向量运算 `dot_product()` 和 `scale_and_add()` 替换为循环。测试这个程序, 并添加必要的注释, 提高代码的可读性。
7. 重写高斯消去法程序, 不使用 `Matrix` 库, 只使用内置数组或 `vector`。
8. 动态演示高斯消去法。
9. 重写非成员函数 `apply()`, 返回所应用函数返回类型的数组, 即如果 `f` 的返回类型为 `R`, 则 `apply(f, a)` 返回一个 `Matrix<R>` 对象。注意: 本题的求解要用到本书未涉及的一些模板方面的知识。
10. 你所使用的 `rand()` 到底有多随机? 编写程序, 读入两个整数 `n` 和 `d`, 调用 `randint(n)` `d` 次, 记录生成的随机数。输出随机数的分布, 即 `[0: n)` 间每个值出现的次数, 观察计数的相似程度。测试较小的 `n` 和 `d`, 观察生成较少的随机数是否会导致明显的分布不均。
11. 编写与 24.5.3 节中 `swap_rows()` 对应的 `swap_columns()`。显然, 你必须阅读、理解 `Matrix` 库中的一些代码, 才能完成这个函数的设计。不必太在意效率: 让 `swap_columns()` 与 `swap_rows()` 一样快是不可能的。
12. 实现

```
Matrix<double> operator * (Matrix<double>, 2 > &, Matrix<double> > &);
```

和

```
Matrix < double, N > operator + ( Matrix < double, N > &, Matrix < double, N > & );
```

如果需要,请在相关教科书中寻找相关的数学定义。

附言

如果你不太喜欢数学,那么你可能也不太喜欢这一章,你可能应该选择一些不需要本章知识的工作。另一方面,如果你喜欢数学,我们希望你能仔细体会基本的数学概念和代码实现之间是如此接近。

“‘不安全’就意味着‘有人可能付出生命代价’。”

本章介绍嵌入式程序设计，即介绍为“小设备”编写程序的基本知识，而不是为那些配置有屏幕和键盘的传统计算机编写程序。我们重点讨论编写“更接近硬件”的程序所需的基本原理、程序设计技术、语言特性和编码规范。语言方面主要包括资源管理、内存管理、指针和数组的使用以及位运算等问题，重点是低层特性的使用和替代方法。我们不会介绍特殊的机器架构或者直接访问硬件设备的方法，这些应该是专门文档和手册介绍的内容。本章最后会给出一个加密/解密算法的实现例子。

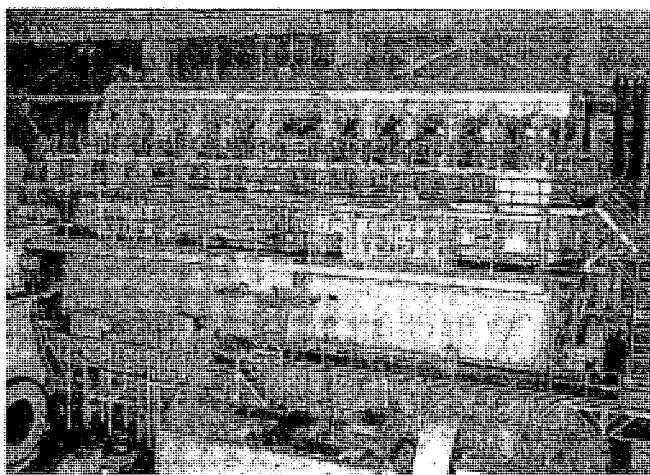
25.1 嵌入式系统

实际上，世界上大多数的计算机系统，都不太像“计算机”。它们可能是一个大型系统的组成部分，或者仅仅是一个“小设备”。例如：

- 汽车：一台新式汽车可能配有数十台计算机，用于控制燃油喷射、监控引擎性能、调节收音机、控制刹车、监控轮胎充气不足的情况、控制风挡雨刷等。
- 电话：一部新式手机内至少有两台计算机，通常其中一台专门用于信号处理。
- 飞机：一架现代飞机内也有多台计算机，完成从运行乘客娱乐系统到摆动翼端优化飞行特性等各种各样的任务。
- 照相机：现在已经有配置 5 个以上处理器的照相机了，甚至每个镜头都由独立的处理器来控制。
- 信用卡（“智能卡”的一种）。
- 医疗设备监测器和控制器（例如 CAT 扫描仪）。
- 电梯（升降机）。
- PDA (Personal Digital Assistant, 个人数字助理)。
- 打印机控制器。
- 音响系统。
- MP3 播放器。
- 厨房用具（如电饭煲和烤面包机）。
- 电话交换设备（通常包含数千个专用计算机）。
- 水泵控制器（抽水或者抽油等）。
- 焊接机器人：在人类焊工无法进入的狭小或者危险的环境中完成焊接任务。
- 风力涡轮机：一些风力涡轮机高达 70 米（210 英尺），能产生数兆瓦电能。
- 防潮闸控制器。
- 装配线质量监控器。
- 条码阅读器。

- 汽车组装机器人。
- 离心机控制器(很多医学分析过程中要用到)。
- 磁盘驱动器控制器。

这些计算机都是大型系统的一部分。这些“大型系统”通常看起来不像一台计算机,我们通常也不把它们看做计算机。当看到一辆小轿车沿着大街驶来,我们绝会说:“快看!那儿有一个分布式计算机系统!”是的,小轿车也是一个分布式计算机系统,但其运行已经与机械系统、电子系统以及电气系统非常紧密地结合在一起了,我们实际上无法孤立地考察计算机系统。它在计算上(时间上和空间上)的限制和程序正确性的定义上都已经不能与整个系统分开了。通常,一台嵌入式计算机控制某个物理设备,计算机的正确行为被定义为物理设备的正常操作。我们来看一台大型的船用柴油机,如下图所示:



注意位于5号汽缸前端的人。这是一台巨大的引擎,这种引擎为大型船舶提供动力。如果一台这样的引擎发生故障,你就会在早报的头版上看到相关的新闻。在这个引擎上,每个汽缸前端都有一个由三台计算机组成的汽缸控制系统。每个汽缸控制系统都通过两个独立的网络系统和引擎控制系统(由另外三台计算机组成)相连。引擎控制系统又连接到控制室,在那里,机械工程师可以通过一个专门的GUI系统与引擎控制系统交互。在航线中心,可以使用无线电系统(通过卫星)对整个系统进行远程监控。更多的实例可参考第1章。

那么,从一个程序员的角度来看,运行在这样一台引擎内的计算机之上的程序有什么特殊之处呢?更一般地,为各种各样的嵌入式系统编写程序时,有哪些问题是原来编写“普通程序”时不必过于担心,但现在需要特别关注的呢?

- 通常,在嵌入式系统中可靠性(reliability)是至关重要的:因为故障可能是突如其来的、损失巨大的(可能“达到数十亿美元”)而且可能是致命的(如船舶失事时船上的人员或者类似环境下的动物)。
- 通常,在嵌入式系统中资源(内存、处理器、能源等)是有限的:对于引擎中的计算机,这可能不是一个问题。但请考虑手机、传感器、PDA、航天探测器等系统,其中的资源问题就很严重了。在我们的日常应用中,配置主频2 GHz的双核CPU和2GB内存的笔记本很常见,但飞机上或航天探测器中的关键计算机系统可能只配备60 MHz的处理器和256 KB的内存,而一个小型装置中的计算机系统可能只有主频低于1 MHz的处理器和几百个字

的内存。能够抵抗环境灾难(如振动、碰撞、不稳定的电力供应、温度过高过低、湿度过高、人为破坏等)的计算机通常比普通的笔记本慢很多。

- 通常,在嵌入式系统中实时响应(real-time response)是必需的:如果燃油喷射器错过了一个喷射周期,就意味着一个能输出十万马力的非常复杂的系统会发生糟糕的事情。错过几个周期,即不能正常工作一秒钟左右,推进器就会产生奇怪的行为——可能产生 33 英尺(10 米)的距离偏差和 130 吨的动力偏差。显然你不希望发生这样的事情。
- 通常,嵌入式系统需要一年到头不间断地正常运行:或许计算机系统是工作在绕地球轨道运行的通信卫星上;又或许系统非常便宜,因而制造量极为巨大,较高的返修率会给厂商带来极大损失(如 MP3 播放器、带嵌入式芯片的信用卡以及汽车的燃油喷射器)。在美国,电话骨干网交换机的强制可靠性标准是 20 年中停机时间在 20 分钟以内(这甚至没有考虑更新交换机程序所需的停机时间)。
- 通常,对于嵌入式系统,手工维护(hands-on maintenance)是不可行的或者非常少见:对于一艘大型船舶,大概每两年左右进港一次进行整体维护,这时你就可以进行计算机系统的维护了,但前提是计算机专家此时恰好有时间、又恰好在船舶停靠地。不定期的人工维护是不可行的,例如,当船舶在太平洋中央遇到大风暴时,是不允许出现任何故障的。再如,你是不可能派人去维修绕火星飞行的航天探测器的。

很少有系统面临所有这些问题,但即使仅仅面临其中一个问题,也需要领域专家来解决。本章的目标不是使你立刻成为专家,设定这样的目标是很愚蠢也是很不负责任的。我们的目标是使你了解基本问题和解决问题的基本概念,使你对构造这类系统的基本技术有所体会。也许经过学习后,你就会对这些重要的技术产生兴趣,产生深入学习的愿望。设计和实现嵌入式系统的人,对我们科技文明的很多方面都相当重要。

那么本章内容与初学者有关吗?与 C++ 程序员有关吗?回答是肯定的。现实生活中,嵌入式系统的数量远远多于传统 PC。我们的程序设计工作中,有一大部分与嵌入式系统有关,因而你的第一个实际工作就可能涉及嵌入式系统程序设计。而且,本节开头列出的嵌入式系统的例子,都是我本人亲眼所见的使用 C++ 进行程序设计的实际例子。

25.2 基本概念

嵌入式系统程序设计工作中的很大一部分与普通程序设计没有太大区别,因此本书介绍的大部分概念和技术仍然适用。不过,本章的重点是描述两者之间的不同之处:我们必须调整程序设计语言工具的使用方式,以适应嵌入式系统的一些限制,而且通常我们需要以底层方式访问硬件:

- 正确性(correctness):对于嵌入式系统,正确性比普通系统更为重要。“正确性”不只是一个抽象概念。在嵌入式系统中,一个程序的正确性不仅仅是一个产生正确结果的问题,还意味着要在正确的时间得到正确的结果以及只使用允许范围内的资源。理想情况下,我们要小心、仔细地定义正确性包含哪些内容,但通常,完整的定义只有在试验之后才能得。一般来说,整个系统都实现完毕后(不仅包括计算机系统的实现,还包含物理设备等其他部分)才能进行关键性的实验。完整的正确性定义对于嵌入式系统来说既非常困难又非常重要。“非常困难”是指“给定的时间和可用的资源不足以完成”,我们必须竭尽所能使用所有可用的工具和技术。幸运的是,在某个特定领域,可运用的规范、仿真方法、测

试技术和其他技术可能超乎我们的想象,有效使用的话可能帮助我们完成目标。“非常重要”是指“故障会导致极大的损害甚至毁灭性的后果”。

- **容错(fault tolerance)**: 我们必须仔细定义程序应该处理哪些情况。例如,对于一个普通的学生练习程序,如果我们在程序演示时踢掉电线,还要求它能继续正常工作,显然是不公平的。对于一个普通的PC应用程序,不应该要求它能处理电源故障。但是,对于嵌入式系统,电源故障并不罕见,在某些情况下程序应该有能力对此进行处理。例如,系统的关键部件可能配备双电源、备用电池等。“我假定硬件正常工作”不应成为应用程序不进行错误处理的借口。因为,经过很长时间运行,面对各种各样的工作条件,硬件发生故障是很正常的。例如,一些电话交换机程序和一些航天器程序会假设计算机内存中的值迟早会发生位偏转(例如,从0变成1)。或者,可能内存中某个位一直保持为1,将其改变为0的操作都会被忽略掉。如果内存足够大,而且使用时间较长的话,这类错误总是会发生。如果内存暴露于强辐射中,例如系统运行于地球大气层之外,这种错误就会很快发生。当我们设计一个系统时(无论是不是嵌入式系统),应该明确系统应提供的容错能力。通常的默认情况是假定硬件会按规定方式工作,对于更重要的系统,这个假设就需要调整了。
- **不停机(no downtime)**: 嵌入式系统一般需要长时间运行,其间不进行软件更新,也无需有经验的了解系统实现的操作员人工干预。“长时间”可以是几天、几个月、几年甚至硬件的整个生命周期。其他类型的系统可能也有这样的需求,但嵌入式系统与大量“普通应用”和本书中的示例程序、系统程序有着非常大的不同。这种“必须永远运行”的需求意味着对错误处理和资源管理的极高要求。那“资源”又是什么呢?所谓资源就是机器只能提供有限数量的那些东西。在程序中,你需要显式地获取资源(“申请资源”、“分配”),使用完毕后还应该将其归还系统(“释放”——“release”、“free”、“deallocate”,可以显式或隐式归还)。资源的例子很多,如内存、文件句柄、网络连接(套接字)和锁等。对于长期运行的程序,除了一些需要一直使用的资源之外,其他资源在使用完毕后都必须释放。例如,如果一个程序每天都忘记关闭一个文件,会使大多数系统在大约一个月后崩溃。如果一个程序每天都忘记释放100字节的内存,那么一年中就会浪费大约32KB内存——这足以令一个小型设备在几个月后崩溃。这种资源“泄漏”问题最令人讨厌的是,程序会良好运行几个月,然后突然崩溃。如果程序必然崩溃,那么我们宁愿它尽可能早地崩溃,以便我们及时发现、修正错误。我们希望系统能在提交用户之前就早早地暴露问题,而不是在用户使用过程中出现错误。
- **实时性限制(real-time constraints)**: 对于一个嵌入式系统,如果每个操作都严格要求在一个时限之前完成,那么我们称它是硬实时(hard real time)的。如果大多数时间要求操作在时限内完成,但对于偶尔的超时能够忍受,那么就称为软实时(soft real time)系统。汽车车窗控制器和立体声音响放大器都属于软实时系统:人们不会注意到车窗的移动延迟了零点几秒钟;只有受过训练的人才会察觉到音高变化时几毫秒的延迟。一个硬实时系统的例子是燃油喷射器,喷射燃油的时间必须严格地与活塞运动同步。如果时间偏差哪怕零点几毫秒,引擎性能就会受影响,磨损也会更严重。如果时间偏差更大的话,甚至会使引擎停止工作,从而导致一起事故甚至灾难。
- **可预测性(predictability)**: 对于嵌入式系统程序来说,可预测性是一个关键概念。显然,这个术语有很多直观的含义,但对于嵌入式系统程序设计,它有一个专门的定义:如果一个

操作在一台给定的计算机上每次的执行时间总是相同的,而同类操作的执行时间也都是相同的,那么我们就称这个操作是可预测的。例如,若 x 和 y 是整数, $x+y$ 的执行时间是不变的,而 $xx+yy$ (xx 和 yy 也是整数) 的执行时间与 $x+y$ 也总是相同的。通常,我们可以忽略由系统架构造成的细微的运行时间差异(如高速缓存和流水线造成的运行时间差异),操作的运行时间就取其上限。在硬实时系统中,绝对不能使用不可预测的操作,在软实时系统中可以使用,但也必须非常小心。一个经典的不可预测操作的例子是列表的顺序搜索(如 `find()`),列表的元素数目是未知的,也很难给出其上限。只有当我们能确切地预测列表元素数目或者至少是能预测最大元素数目时,顺序搜索才能用于硬实时系统。也就是说,为了保证请求在给定时限内被响应,我们必须能够(也许需要借助于代码分析工具)计算所有在时限之前可能执行的代码流的执行时间。

- 并发性(concurrency): 一个嵌入式系统通常需要响应来自于外部的事件。因而程序可能需要同时处理很多事情,因为有可能很多外部事件同时发生。程序同时处理多个动作,称为并发(concurrent)或并行(parallel)。不幸的是,那些吸引人的、有难度的、重要的并行程序设计知识已经超出了本书的讨论范围。

25.2.1 可预测性

C++ 的可预测性相当好,但还不够完美。除了以下几个语言特性外,所有其他 C++ 语言特性(包括虚函数调用)都是可预测的:

- 动态内存空间分配 `new` 和 `delete` (参见 25.3 节)
- 异常(参见 19.5 节)
- 动态类型转换 `dynamic_cast` (参见附录 A.5.7)

应该避免在硬实时系统中使用这些语言特性。我们将在 25.3 节详细讨论 `new` 和 `delete` 所存在的问题,这是一个根本性的问题,任何语言实现都会存在。注意,标准库中的 `string` 和标准容器(`vector`、`map` 等)间接地使用了动态内存分配,因此它们也是不可预测的。`dynamic_cast` 的问题则是当前实现所导致的,而非根本性问题。

异常的问题在于,对每个 `throw`,如果不考察更大范围的代码,程序员无法知道需要花费多长时间才能找到与之匹配的 `catch`,甚至是否存在这样一个 `catch` 都无法获知。在一个嵌入式系统程序中,最好期盼确实存在这样一个 `catch`,而且在抛出异常后能及时执行到这个 `catch`。因为我们不能只依赖调试工具的 C++ 程序员发现这类问题。如果有这么一个工具,它能找到每个 `throw` 所匹配的 `catch`,并能计算出多长时间能到达 `catch`,就能解决异常所面临的这个问题了。但到目前为止,这样的工具还处于研究阶段,离实用还很遥远。因此,如果程序必须是可预测的,你就需要使用返回代码等老式技术来编写错误处理程序,虽然这类技术可能冗长乏味,但它们是可预测的。

25.2.2 理想

编写嵌入式系统程序的过程中存在这样一种危险:对性能和可靠性的追求导致程序员倒退到只使用低层语言特性的地步。如果是编写一小段程序,这样做还是可行的。但是,一般情况下,它容易使整体设计陷入混乱,使程序的正确性难以验证,还会大大增加系统开发的成本和时间。

与以往一样,我们的理想是尽量使用较高层次的抽象,以便能够很好地描述要解决的问题。不要退回到编写汇编代码的地步!与以往一样,尽可能直接用代码表达你的思想(已给定的所有条件)。与以往一样,努力编写最清晰、最干净、最易维护的代码。除非真的需要,否则不要执着

于代码优化。性能(时间或空间)对一个嵌入式系统通常很重要,但是试图榨干每一小段代码的性能极限就是误入歧途了。而且,对于很多嵌入式系统而言,重要的是正确性和“足够快”。超越“足够快”就没有意义了,系统只能空闲下来,等待进行下一个操作。在编写每一小段代码时都试图达到最高效率,既花费大量时间,又会导致大量 bug,而且通常还会导致失去优化的机会,因为这样实现的算法和数据结构难以理解、难以修改。例如,这种“低层优化”编程方式通常会导致内存优化难以进行,因为大量相似的代码片段出现在很多地方,但又无法共享这些代码,因为它们都有细微的差异。

John Bentley(以设计高效代码著称)提出了两条“优化法则”:

- 第一法则: 不要做优化。
- 第二法则(仅对行家里手): 还是不要做优化。

在进行优化之前,必须确认你完全理解了系统。唯有这样,你才能确信优化是正确而又可靠的。在程序设计过程中,应该把精力集中在算法和数据结构上。当系统的早期版本可以运行起来后,再根据需要仔细测试、调节系统。幸运的是,这种朴素的程序设计策略也可能带来惊喜: 简洁的代码有时会足够快而且不会占用太多的内存。即便有这种可能,也不要报太大期望,相反的情况也是很常见的,还是要进行仔细的测试。

25.2.3 生活在故障中

设想我们准备设计并实现一个不会失效的系统。这里“不会失效”的意思是“可以在没有人工干预的情况下正常工作一个月”。那么我们必须防御哪些类型的故障呢? 我们当然可以排除太阳向新星演变的情况,系统被大象踩踏的情况应该也无需考虑。但是,一般来说我们很难估计会发生什么样的故障。对于一个特定系统,我们可以也应该假定哪些故障更容易发生,例如:

- 功率骤变/电源故障
- 插头从插座上脱落
- 系统被落下的碎片击中,处理器被损坏
- 系统坠落(硬盘可能会因为冲击而损坏)
- X 射线导致内存中某些位的值不按程序语言的定义而改变

瞬时故障通常是最难查找的,所谓瞬时故障(transient error),就是指在“某些时候”会发生,但不会在程序每次运行时都发生的故障。例如,我们听说过处理器只有在温度超过 130 华氏度(54 摄氏度)时才会行为异常,正常情况下是不会达到这么高的温度的。但是,如果系统(偶然地、不小心地)堆积在工厂车间的角落里,散热不好的话,还是有可能达到这个温度的,当然系统在实验室中进行测试时不会达到这个温度。

在实验室之外发生的故障是很难修复的。你很难想象,为了让 JPL 的工程师能够监测火星巡回者号上的软件和硬件故障,并能在弄清问题后通过软件更新的方式来修复故障,在设计和实现系统时会花费多么大的努力。

为了设计和实现一个具有容错能力的系统,领域知识(即关于系统本身、它的工作环境及其使用方式的知识)是必需的。在本章中,我们只能涉及一些一般原则。注意,这里讨论的每条“一般原则”都是一个庞大的主题,都有几十年的研究和开发历史,相关的文献都数以千计。

- 避免资源泄漏: 绝不能发生泄漏。要明确程序使用哪些资源,要确保你(完全)拥有这些资源。任何泄漏最终都会令系统或者子系统崩溃,最重要的资源是 CPU 时间和内存。通常,程序还会使用其他资源,如锁、通信信道、文件等。

- 复制：如果某个硬件资源(如计算机、输出设备、轮子等)的正常运转对系统至关重要，那么设计者就面临这样一个基本的选择——是否应该为关键资源配置备份？对于硬件故障，我们要么简单地承受故障，要么配置热备设备，在故障时通过软件切换到热备设备。例如，船用柴油机燃油喷射器的控制器有三重备份，备份之间通过一个双重备份的网络链接。注意，“热备”设备不需要与原设备完全一样(例如，可能航天探测器的主天线接收能力很强，而备份天线较弱)。而且，在系统无故障时，“热备”设备通常也可以投入工作，以提升系统性能。
- 自检测：要了解掌握程序(或硬件)什么时候出现故障。硬件设备(如存储设备)通常都能监测自身的运行状况，对小故障进行修复，将无法处理的严重故障报告给用户，这对于我们进行故障检测非常有帮助。软件则可以检查数据结构的一致性，检查不变量(参见 9.4.3 节)，以及依赖内部的“完整性检查”(断言)进行故障检测。不幸的是，自检测机制本身也有可能是不可靠的，报告错误的过程本身可能导致一个新的错误，对这种情况必须加以小心——对错误检测模块本身的完全彻底的检测是非常困难的。
- 能够迅速离开有错误的代码：解决的策略就是系统的模块化。每个模块都完成一项特定的工作，在此之上完成基于模块的错误处理。如果一个模块无法完成自己的工作，它可以将这一情况报告给其他模块。保持模块内的错误处理尽量简单(这样，故障恢复的可能性就更高，修复效率也更高)，由其他模块负责更严重的错误。一个高可靠的系统一定是模块化的和层次化的。在每个层次中，严重错误都报告给下一层次来处理。最终的层次，可能是由操作人员来处理。一个模块收到一个严重错误(另一个模块无法自己处理的错误)的通知后，可以采取适当的措施，可以重启错误模块，或者启动一个更简单(但也更可靠)的“备份”模块。对于一个给定系统，准确定义什么是“模块”，应该是系统整体设计的一部分，但你可以把模块看做一个类、一个库、一个程序或者一台计算机上的所有程序。
- 监控对子系统——如果子系统自身不能或没有监测自身故障的话。在一个多层系统中，上层模块可以监控下层模块。很多不允许失效的系统(如船用引擎或者空间站的控制器)对关键子系统都配置三重备份。这种三重备份不仅仅是为了设置两个热备设备，还有一个很重要的目的：在设备行为不一致时，通过投票，采用少数服从多数的策略(一个服从两个)来确定正确的结果。在多层结构很难实施的地方(即系统的最高层，或者不允许失效的子系统)，三重备份就显得非常有用了。

我们可以设计更多这样的原则，并在实现中小心保证，但系统仍然会出现不可预知的问题。因此，在交付用户使用之前，还是需要进行系统的、全面的测试，参见第 26 章。

25.3 内存管理

计算机中两种最重要的资源是时间(执行指令)和空间(保存数据和程序的内存)。在 C++ 中，有三种分配内存的方法(参见 17.4 节和附录 A.4.2)：

- 静态内存：是由连接器分配的，其生命期为整个程序的运行期间。
- 栈内存(也称为自动内存)：在调用函数时分配，当函数返回时释放。
- 动态内存(也称为堆)：用 new 操作分配，用 delete 操作释放。

下面,我们从嵌入式程序设计的角度来考察这几种内存分配方式。特别地,我们将把可预测性(参见 25.2.1 节)作为必备的性质考虑在内,也就是说,我们针对的是硬实时系统程序设计和安全系统程序设计。

在嵌入式系统程序设计中,静态内存分配不会引起任何特殊的问题:因为所有的内存分配工作都在程序开始运行之前就已经完成了,也远在系统部署之前。

栈内存如果分配过多,就可能导致一些问题,但这并不难处理。系统设计者须确保没有任何程序在执行时会使栈溢出,这通常意味着函数调用的最深层次不能超过限制,即我们必须保证调用链不会太长(例如, f_1 调用 f_2 , f_2 调用 f_3 , ..., 调用 f_n)。在某些系统中,可能就需要禁止使用递归函数了。这对某些系统和某些递归函数是合理的,但并不是对所有系统和递归函数都如此。例如,我们知道 $\text{factorial}(10)$ 最多产生对 factorial 的 10 层调用,因此可以很容易确保栈不会溢出。但是,嵌入式系统程序员可能更愿意使用循环来实现 factorial (参见 15.5 节),以避免任何疑问或意外。

在嵌入式程序中,动态内存分配通常是被禁止或者受到严格限制的,即 `new` 或者被禁止,或者只在启动时使用,而 `delete` 则被严格禁止。基本的原因是:

- 可预测性:动态内存分配是不可预测的,也就是说,它不能保证在固定时间内完成。实际上,不能按时完成的情况还不是少数,因为许多 `new` 的实现都有这样一个特点:在已经分配和释放了很多对象后,再分配新的对象,所花费的时间会呈上升趋势。
- 碎片(fragmentation):动态内存分配会造成碎片问题,即在分配和释放了大量对象后,剩余的内存会“碎片化”——空闲内存被分割成大量小“空洞”,每个空洞都很小,无法容纳程序所需对象,从而使这些空闲内存毫无用处。因此,可用空闲内存量远远小于初始内存总量减去已分配的内存量。

下一节会解释为什么会出现这种不可接受的情况。重要的是在硬实时程序设计和安全程序设计中,我们必须避免使用 `new` 和 `delete`。下面几节会介绍一些方法,可以使用栈和存储池技术系统地避免动态内存分配带来的问题。

25.3.1 动态内存分配存在的问题

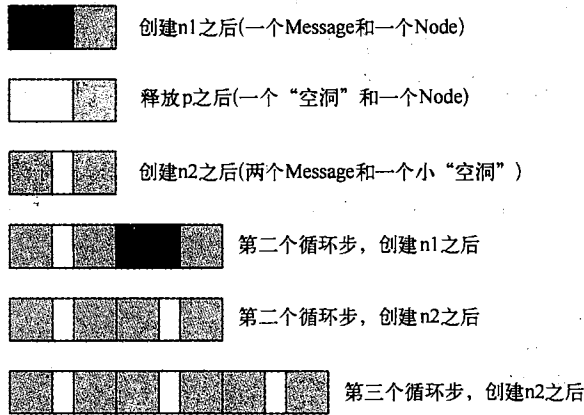
`new` 的问题究竟在哪里呢?实际上问题是出在 `new` 和 `delete` 的结合使用上。观察下面程序中内存分配和释放的过程:

```
Message* get_input(Device&);           // make a Message on the free store

while(/* ... */) {
    Message* p = get_input(dev);
    // ...
    Node* n1 = new Node(arg1,arg2);
    // ...
    delete p;
    Node* n2 = new Node (arg3,arg4);
    // ...
}
```

在每个循环步中,我们创建了两个 `Node`,在此期间,我们还分配了一个 `Message`,然后又释放了它。当我们需要用某个“设备”而来的输入创建一个数据结构时,常常会使用这样的代码。看看这段代码,每执行一个循环步,我们可能期望“消耗” $2 * \text{sizeof}(\text{Node})$ 个字节的内存(再加上动态内存分配的额外开销)。但不幸的是,真正的内存“消耗”并不一定如我们所愿。实际上,每个循环步总是会消耗掉更多的内存。

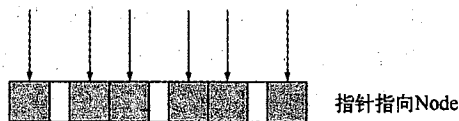
我们假定系统使用一个简单(但并非与实际不符)的内存管理程序。另外假定一个 Message 比一个 Node 稍大。下图展示了动态内存的使用情况,其中 Message 用黑色表示,Node 用灰色表示,而白色表示“空洞”(即“未使用空间”):



由此可见,每执行一个循环步,我们就会在动态内存中留下一些未用空间(“空洞”)。这些空洞可能只有几个字节大小,但如果我们不能加以有效利用,其危害与内存泄漏是一样的——即使是微小的泄漏,在长时间运行后也会导致系统崩溃。在内存中,空闲空间分散,形成很多小“空洞”,无法满足新的内存需求的情况,就称为内存碎片。内存管理程序最终会把足够大的“空洞”用尽,只留下无法使用的小空洞。这是任何频繁使用 new 和 delete 的系统长期运行后都会遇到的一个严重问题,最终,内存中布满了无法使用的碎片。此时再执行 new 操作,就需要在大量对象和碎片中搜索足够大的区域,所花费的时间就会急剧增加。显然,这对于嵌入式系统来说是不可接受的。对于非嵌入式系统,这也是一个严重问题。

为什么不“语言”或“系统”来处理这个问题呢?或者,我们为什么不能编写不会形成“空洞”的程序呢?我们首先看一下消除“空洞”的最直接的方法:移动 Node,将空闲空间压缩成一片连续的区域,这样就可以用来存储新的对象了。

不幸的是,“系统”无法完成这样的任务。原因在于 C++ 是直接用内存地址来访问对象的。例如,指针 n1 和 n2 中都是对象的实际内存地址。因此,如果我们移动了对象,这些指针就不再指向正确的对象,其中的地址就变为无效了。下图给出了指针保存对象地址的示意:



现在,我们如果移动对象,整理碎片,就会出现下面的情况:



不幸的是,由于移动对象时没有相应地修改指针,现在的指针已经乱七八糟了。那么我们为什么

不在移动对象的同时修改指针呢？我们可以编写一个程序来完成这个工作，但是前提是必须知道数据结构的细节。一般情况下，“系统”（C++ 运行时支持系统）是无法知道指针在哪里的，也就是说，给定一个对象，无法回答“程序中的哪些指针现在指向这个对象”。即使可以回答这个问题，这种方法（称为压缩垃圾收集，compacting garbage collection）通常也不是最好的方法。例如，为了完成收集任务，除了程序所使用的内存外，还需要两倍于此的空间来跟踪指针以及移动对象。在嵌入式系统中，是不会有这么多额外内存空间的。另外，一个高效的垃圾内存收集程序很难达到可预测性。

我们自己当然可以回答“指针在哪”的问题，因此可以自己编写程序来进行空间压缩。这是可行的，但更简单的方法是从根本上避免碎片的出现。在本例中，我们可以在分配 Message 之前为两个 Node 分配空间：

```
while(...){
    Node* n1 = new Node;
    Node* n2 = new Node;
    Message* p = get_input(dev);
    // ... store information in nodes ...
    delete p;
    // ...
}
```

但是，用重整代码的方法来避免碎片问题通常比较困难。最乐观地估计，既避免碎片，同时又要保证代码仍旧可靠，也是一项非常困难的工作。而且，代码的重整可能与其他编码基本原则冲突。因此，我们倾向于限制动态内存分配的使用，这样就从根本上消除了碎片问题。通常，预防比治疗更有效。

试一试 将上面的程序补充完整，输出创建的对象的地​​址和大小，观察是否会出现“空洞”，“空洞”又是如何分布的。如果有时间的话，试着画一下内存布局（就像前面那几个图一样），这能帮你更好地理解这个问题。

25.3.2 动态内存分配的替代方法

我们已经决定从根本上避免碎片，但如何做到呢？首先，一个简单的事实是：单独使用 new 操作不会导致碎片，用 delete 操作释放内存时才会产生空洞。因此，我们第一步先禁止 delete。这意味着，一旦为对象分配了内存空间，那么其生命周期就会持续到程序结束。

如果不再使用 delete 了，new 就是可预测的了么？也就是说，所有 new 操作都会花费相同的时间了么？对于一般的实现，确实是这样，但并不是所有系统都保证如此。通常，嵌入式系统中都有一段初始化代码，在加电或重启后完成初始化任务。在初始化期间，我们可以任意分配内存空间，只要不超过限额即可。分配方式可以使用 new，也可以使用全局（静态）内存。从程序结构的角度来说，应该尽量避免使用全局数据，但全局内存分配方式可以实现内存空间的预分配。在这方面更准确的规则应作为系统编程规范的一部分（参见 25.6 节）

有以下两种数据结构在实现可预测内存分配时非常有用：

- **栈**：在栈中，你可以分配任意大小的内存空间（分配的总量不超过预设的最大值），最后分配的空间总是最先被释放。也就是说，栈只在栈顶一端增长和缩小。因而也就不存在碎片问题，因为内存空间的分配和释放是不会交叉的。
- **存储池**：所谓存储池，就是一组相同大小的对象的集合。只要需分配的对象数未超过存储池的容量，就可以在其中任意分配和释放对象。由于所有对象都是相同大小，因此也不会产生碎片。

采用栈和存储池，分配和释放都是可预测的，速度也很快。

这样，对于硬实时系统或者关键系统，我们可以视需要自己定义栈和存储池。但最好能使用

现成的、已经过测试的栈和存储池代码(只要其定义符合我们的需要就可以使用)。

注意,我们不能使用 C++ 标准库中的容器(vector、map 等)和 string,因为它们间接使用了 new。你可以创建(购买或借用)可预测的“类标准”容器,当然这些代码的用途不仅仅局限于嵌入式系统。

注意,嵌入式系统通常在可靠性上要求非常严格,因此,无论选择怎样的解决方案,我们都不能倒退到直接使用大量低层特性的程序设计风格。充斥着指针、显式类型转换等特性的代码,其正确性是极难保证的。

25.3.3 存储池实例

存储池是这样一种数据结构,我们可从中分配指定类型的对象,随后可将这些对象释放。下图说明了存储池的工作原理,其中灰色表示“已分配的对象”,而白色表示“空闲空间”:



我们可以定义 Pool 如下:

```
template<class T, int N>class Pool {    // Pool of N objects of type T
public:
    Pool();                // make pool of N Ts
    T* get();              // get a T from the pool; return 0 if no free Ts
    void free(T*);         // return a T given out by get() to the pool
    int available() const; // number of free Ts
private:
    // space for T[N] and data to keep track of which Ts are allocated
    // and which are not (e.g., a list of free objects)
};
```

每个 Pool 对象都包含类型相同的一组对象,对象的数目有上限值。Pool 的使用方法如下面代码所示:

```
Pool<Small_buffer,10> sb_pool;
Pool<Status_indicator,200> indicator_pool;

Small_buffer* p = sb_pool.get();
// ...
sb_pool.free(p);
```

程序员应该确保存储池不会被耗尽,“确保”的准确含义依赖于具体应用。对于某些系统,程序员应该保证只有在存储池有空闲空间的情况下才调用 get()。在另外一些系统中,程序员可以检测 get() 的返回值,在返回值为 0 的情况下进行一些补救措施。第二种策略的一个典型例子是电话系统,假定它最多同时处理 100 000 个呼叫。每个呼叫都需要一些资源,比如一个拨号缓冲区。如果系统中的拨号缓冲区都已耗尽(如 dial_buffer_pool.get() 返回 0),系统可以拒绝建立新的通话连接(还可能“杀掉”一些已有的通话连接来释放一些空间)。拨打电话的人则可以稍后再拨。

当然,我们的 Pool 模板仅仅是存储池一般思想的一种实现,还可以根据需要进行其他实现方式。例如,对于内存分配限制不那么苛刻的系统,我们可以修改存储池的定义,在构造函数中指定元素数目,甚至在需要时改变元素数目。

25.3.4 栈实例

栈是这样一种数据结构,我们可以从中分配内存空间,而最后分配的区域被最先释放。下图说明了栈的工作方式,其中灰色表示“已分配内存”,白色表示“空闲空间”:



如上图所示，栈向右“生长”。

定义一个对象栈，与定义一个对象存储池类似：

```
template<class T, int N> class Stack {    // stack of Ts
    // ...
};
```

然而，大多数系统都需要为不同大小的对象分配内存。存储池无法满足这种需求，但栈却可以。下面我们就展示如何定义一个能从中分配大小不同的“原始”内存空间，而非固定大小对象的栈。

```
template<int N> class Stack {    // stack of N bytes
public:
    Stack();                    // make an N-byte stack
    void* get(int n);           // allocate n bytes from the stack;
                                // return 0 if no free space
    void free();                // return the last value returned by get() to the stack
    int available() const;      // number of available bytes
private:
    // space for char[N] and data to keep track of what is allocated
    // and what is not (e.g., a top-of-stack pointer)
};
```

get() 从栈中分配指定大小的内存空间，返回指向起始地址的 void* 指针，因此，我们需要显式将其转换为所需的类型。这种栈的使用方式如下：

```
Stack<50*1024> my_free_store; // 50K worth of storage to be used as a stack
```

```
void* pv1 = my_free_store.get(1024);
int* buffer = static_cast<int*>(pv1);
```

```
void* pv2 = my_free_store.get(sizeof(Connection));
Connection* pconn = new(pv2) Connection(incoming,outgoing,buffer);
```

static_cast 的使用已经在 17.8 节中介绍过了。语法 new(pv2) 表示“定址 new”，即“在 pv2 指向的内存空间中创建一个对象”。也就是说，它并不分配新的内存空间。这段代码假定 Connection 有一个构造函数，接受参数 (incoming, outgoing, buffer)。如果没有定义这样的构造函数，编译会失败。

自然地，Stack 模板也只是栈的一般思想的一种实现而已，还可以有其他的实现方式。例如，如果内存分配的限制不那么苛刻，我们可以修改栈的定义，实现在构造函数中指定预分配的空间大小。

25.4 地址、指针和数组

可预测性只是某些嵌入式系统的需求，而可靠性则是所有嵌入式系统都需要的。因此，应该避免使用那些已被证明容易出错的语言特性和程序设计技术（这里是指在嵌入式系统中容易出错，在其他环境中并不一定）。指针就是这样一种语言特性，使用不慎很容易导致错误，有两个问题最为突出：

- （未经检查的和不安全的）显式类型转换
- 将指向数组元素的指针作为参数传递

前一个问题通常可以简单地通过严格禁止使用显式类型转换来解决。指针/数组问题则更微妙,理解起来更有难度,解决方法可以使用(简单的)类或者标准库功能(如 array, 参见 20.9 节)。因此,本节主要讨论如何解决指针/数组问题。

25.4.1 未经检查的类型转换

在低层系统中,物理资源(如外部设备的控制寄存器)及其基础软件通常位于特定的地址。我们不得不在程序中直接使用这些地址,并将它们转换为所需类型:

```
Device_driver* p = reinterpret_cast<Device_driver*>(0xffb8);
```

请参考 17.8 节。这种语法很不常用,你可能需要借助手册和联机帮助才不会写错。硬件资源(资源的寄存器的地址——通常表示为十六进制整数)和指向硬件资源控制软件的指针之间的对应关系是脆弱的。你需要保证其正确性,但又得不到编译器的帮助(因为这本来就不是程序设计语言方面的问题)。通常, int 类型到指针类型的简单转换(reinterpret_cast),是连接一个应用程序和它的重要硬件资源所必需的。但这样的转换是完全未经检查的,因此很容易出错。

只要显式类型转换(reinterpret_cast, static_cast 等,参见附录 A.5.7)并非必需,就应该避免使用。通常,一些先前使用 C 或者 C 风格 C++ 的程序员喜欢使用这种类型转换,但实际上很多情况下是不必要的。

25.4.2 一个问题:不正常的接口

如上 18.5.1 节所述,一个数组常常作为参数,以指针的形式传递给函数(指针通常指向数组的第一个元素)。这样,数组大小就“丢失”了,从而导致接受参数的函数无法判断数组中共有多少个元素。这个问题是很多微妙而难以修正的 bug 的根源。下面,我们考察一些数组/指针问题的例子,并给出一个替代方法。我们以一个非常差的接口程序(但很不幸,这个例子在实际程序中并不罕见)作为开始,然后尝试改进它:

```
void poor(Shape* p, int sz)    // poor interface design
{
    for (int i = 0; i < sz; ++i) p[i].draw();
}

void f(Shape* q, vector<Circle>& s0)    // very bad code
{
    Polygon s1[10];
    Shape s2[10];
    // initialize
    Shape* p1 = new Rectangle(Point(0,0),Point(10,20));
    poor(&s0[0],s0.size());    // #1 (pass the array from the vector)
    poor(s1,10);               // #2
    poor(s2,20);               // #3
    poor(p1,1);                // #4
    delete p1;
    p1 = 0;
    poor(p1,1);                // #5
    poor(q,max);               // #6
}
```

函数 poor() 是一个设计得很差的接口:它使调用者极易出错,又几乎没有给实现者预防错误的机会。

试一试 在继续阅读之前,尝试找出 f() 中的错误。特别是,对 poor() 的哪次调用会导致程序崩溃?

乍一看,这些对 poor() 的调用没有什么问题,但这些代码正是那种会花费程序员整夜时间来

除错的程序，对高水平工程师来说也会是一场噩梦。

1) 元素类型传递错误，如 `poor(&s0[0], s0.size())`。而且 `s0` 还可能是空的，此时 `&s0[0]` 本身就是错的。

2) 使用了“魔数”：`poor(s1, 10)` (此处是正确的)。这里，元素类型也是错误的。

3) 使用了错误的“魔数”：`poor(s2, 20)` (此处是正确的)。

4) 正确的调用：第一个 `poor(p1, 1)`。

5) 传递了一个空指针：第二个 `poor(p1, 1)`。

6) 可能是正确的：`poor(q, max)`。仅看这个代码片段，不能判断这个调用是否正确。为了判断 `q` 指向的数组是否包含至少 `max` 个元素，必须找到 `q` 和 `max` 的定义并获得程序运行到此处时它们的值。

上述这些错误都很简单，我们并未涉及微妙的算法或数据结构问题。所有问题都出在 `poor()` 的接口上，它包含一个以指针方式传递的数组，这导致了一系列的问题。你可以体会一下，我们所使用的 `p1` 和 `s0` 这种无意义的名字是如何使问题更加模糊不清的。这些名字虽然有助记忆，但容易造成混淆，使得这些错误更难以查找。

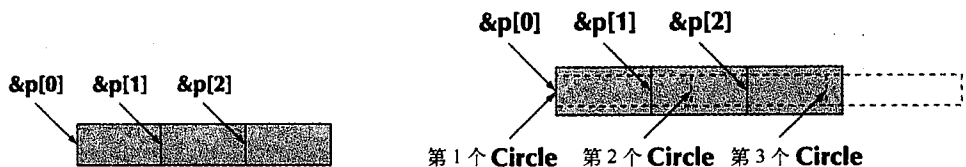
理论上，编译器可以找到其中一些错误(如第二个 `poor(p1, 1)` 中 `p1 == 0` 的情形)。但在现实中，我们之所以能免受这些错误的困扰，主要还是因为编译器发现了程序试图定义抽象类 `Shape` 的对象。但是，这并未解决 `poor()` 接口方面的问题，因此我们无法松口气。接下来，我们将使用一个非抽象的 `Shape`，这样就能专注于接口问题了。

`poor(&s0[0], s0.size())` 到底错在哪里呢？`&s0[0]` 指向一个 `Circle` 数组的首元素，因此它是一个 `Circle*` 指针。`poor` 期待的是一个 `Shape*` 指针，而我们传递给它的是一个 `Shape` 派生类对象指针(`Circle*`)。这显然是允许的：我们需要这种类型转换，因为面向对象程序设计中常常需要用同一段代码对源于同一基类(本例中是 `Shape`)的不同派生类的对象(参见 14.2 节)进行处理。但是，`poor()` 不仅仅把 `Shape*` 作为一个指针来使用，还将它作为数组使用，通过下标访问其元素：

```
for (int i = 0; i < sz; ++i) p[i].draw();
```

这段代码顺序访问内存地址 `&p[0]`、`&p[1]`、`&p[2]` 等上的对象如下图所示：

就内存地址而言，这些指针的间距为 `sizeof(Shape)` (参见 17.3.1 节)。但不幸的是，对于此次 `poor()` 的调用，`sizeof(Circle)` 大于 `sizeof(Shape)`，因此内存布局如下图所示：



也就是说，`poor()` 中调用 `draw()` 时，指针实际指向一个 `Circle` 对象的中间！这很可能立即导致程序崩溃。

`poor(s1, 10)` 的问题更为隐蔽。其中使用了“魔数”10，因此我们很容易立刻怀疑它是问题的根源，但实际上这条语句中还隐藏着一个更深层次的问题。我们使用 `Polygon` 数组作为 `poor` 的参数，就不会遇到 `Circle` 数组所面临的那些问题，原因是 `Polygon` 没有在基类 `Shape` 的基础上增加数据成员(而 `Circle` 加入了新的数据成员，参见 13.8 节和 13.12 节)。也就是说，`sizeof(Shape) ==`

sizeof(Polygon), 更一般地讲, Polygon 和 Shape 具有相同的内存布局。换句话说, 我们真的很“幸运”, 对 Polygon 定义的任何微小修改都会导致程序崩溃。因此, 当前的 poor(s1, 10) 可以正常工作, 但它是一个 bug, 迟早会引起程序错误。这条语句毫无疑问是低质量的代码。

上述这些问题都是程序设计法则“‘D 是 B’并不意味着‘D 的容器是 B 的容器’”在实际代码中的体现(参见 19.3.3 节)。例如:

```
class Circle : public Shape { /* ... */};

void fv(vector<Shape>&);
void f(Shape &);

void g(vector<Circle>& vd, Circle & d)
{
    f(d);           // OK: implicit conversion from Circle to Shape
    fv(vd);          // error: no conversion from vector<Circle> to vector<Shape>
}
```

好了, 我们已经知道上述 poor() 的调用是非常糟糕的代码了, 但这种代码会出现在嵌入式程序中吗? 也就是说, 在安全性和性能要求都很高的领域中, 我们会遇到这类问题吗? 我们是否可以简单地把这种代码作为错误的根源, 告诉“普通程序”的程序员“不要在嵌入式程序中使用这种代码”呢? 恐怕还不能这么简单地处理, 因为很多现代的嵌入式系统都严重依赖 GUI, 而 GUI 程序通常采用面向对象程序设计方法开发, 其代码组织很像前面给出的例子。这方面的例子有很多, 如 iPod 的用户界面、一些手机的用户界面以及“小设备”(包括飞机)上操作人员使用的显式界面等。另外一个例子是很多相似设备(例如很多电动机)的控制器可以构成一个典型的类层次。换句话说, 这种代码, 特别是这种函数声明方式, 确实会在嵌入式程序中出现, 我们必须加以考虑。我们需要一种更安全的方法来传递一组数据, 以避免引起上述严重的问题。

因此, 我们不希望数组参数以“指针 + 大小”的方式传递。那么有什么替代方法吗? 最简单的方法是传递容器(如 vector)的引用, 例如:

```
void poor(Shape* p, int sz);
```

就不存在先前的函数接口所存在的那些问题:

```
void general(vector<Shape>&);
```

如果在你的开发环境中, std::vector(或者等价的工具)是可用的, 那么在函数接口中就一直使用它, 而不要以指针加大小的方式传递内置数组。

如果你无法使用 vector 或等价的工具, 就会陷入困境, 虽然可以直接使用我们定义的接口类 Array_ref, 但仍旧需要一些复杂的语言特性和技术来编写程序。

25.4.3 解决方案: 接口类

不幸的是, 在很多嵌入式系统中我们都不能使用 std::vector, 因为它依赖动态内存分配。一种解决方法是实现一个特殊的 vector, 更简单的方法是定义一个与 vector 功能相似但又不使用动态内存分配的容器。在给出这个容器的定义之前, 先思考一下我们希望这个容器具有什么功能:

- 它只是内存中对象的一个引用(它不拥有对象、不分配、释放对象)。
- 它“知道”自己的大小(这样就有可能实现范围检查)。
- 它“知道”元素的确切类型(这样它就不会成为类型错误的根源)。
- 传递代价(拷贝)低, 传递方式可以是一个(指针, 数量)对。

- 它不能显式地转换为一个指针。
- 通过接口对象，能容易地描述元素范围的子区域。
- 它和内置数组一样容易使用。

我们只是尽可能地接近“和内置数组一样容易使用”这一目标，实际上也不应完全“一样容易使用”，因为那样就意味着“一样容易引起错误”。

下面给出了接口类的一个定义：

```
template<class T>
class Array_ref {
public:
    Array_ref(T* pp, int s) : p(pp), sz(s) {}

    T& operator[](int n) { return p[n]; }
    const T& operator[](int n) const { return p[n]; }

    bool assign(Array_ref a)
    {
        if (a.sz!=sz) return false;
        for (int i=0; i<sz; ++i) { p[i]=a.p[i]; }
        return true;
    }

    void reset(Array_ref a) { reset(a.p,a.sz); }
    void reset(T* pp, int s) { p=pp; sz=s; }

    int size() const { return sz; }

    // default copy operations:
    //   Array_ref doesn't own any resources
    //   Array_ref has reference semantics
private:
    T* p;
    int sz;
};
```

这个接口类 Array_ref 已经尽可能地简化了：

- 没有定义 push_back() (因为可能需要动态内存分配)，也没有定义 at() (可能需要使用异常机制)。
- Array_ref 本质是一种引用，因此复制操作只复制(p, size)，而不会复制引用的对象。
- 不同的 Array_ref 可以用不同的数组进行初始化，这样它们具有相同的类型，但大小不一样。
- 我们可以使用 reset() 来更新(p, size)的值，这样就可以改变 Array_ref 的大小(很多算法要求指定子区域)。
- 没有定义迭代器接口(如果需要的话，加入迭代器功能很容易)。实际上，Array_ref 本质上很接近由两个迭代器描述的一个范围。

Array_ref 并不拥有元素，也不进行内存管理，它只不过是一种访问及传递元素序列的机制。在这一点上，它与标准库的 array(参见 20.9 节)是不同的。

为了简化 Array_ref 的初始化，我们设计了一些有用的辅助函数：

```
template<class T> Array_ref<T> make_ref(T* pp, int s)
{
    return (pp) ? Array_ref<T>(pp,s) : Array_ref<T>(0,0);
}
```

如果我们用一个指针来初始化 `Array_ref`, 那么就必须显式地提供数组的大小。这显然是 `Array_ref` 的一个弱点, 因为调用者有可能提供错误的大小。而且, 如果调用者传递来的指针是从一个派生类指针隐式转换为基类指针的, 如将 `Polygon[10]` 传递给 `Shape*`, 那么我们在 25.4.2 节中讨论的那个棘手的问题就又出现了。但是, 只要保留这种初始化方式, 这个问题就很难解决, 我们有时只能相信程序员。

前一段代码中我们对空指针进行了检查(因为它通常是错误之源), 我们同样也应提防空 `vector`:

```
template<class T> Array_ref<T> make_ref(vector<T>& v)
{
    return (v.size()) ? Array_ref<T>(&v[0], v.size()) : Array_ref<T>(0, 0);
}
```

这段代码实现了用 `vector` 初始化 `Array_ref`, 虽然在很多 `Array_ref` 的应用场合(嵌入式系统)中并不适宜使用 `vector`。不过, 与适合在嵌入式系统中使用的容器(如基于存储池的容器, 参见 25.3.3 节)相比, `vector` 具有很多相似的特点。

最后的一个辅助函数利用内置数组(编译器知道其大小)来初始化 `Array_ref`:

```
template<class T, int s> Array_ref<T> make_ref(T (&pp)[s])
{
    return Array_ref<T>(pp, s);
}
```

`T(&pp)[s]` 的语法有些奇怪, 它声明了一个引用类型参数 `pp`, `pp` 引用的是一个元素类型为 `T`、元素个数为 `s` 的数组。这样, 就可以使用数组来初始化 `Array_ref` 了(数组大小是已知的)。由于 C++ 不允许声明空数组, 所以这里不必对此进行检测:

```
Polygon ar[0]; // error: no elements
```

有了 `Array_ref` 后, 我们就可以重写 25.4.2 中的例程了:

```
void better(Array_ref<Shape> a)
{
    for (int i = 0; i < a.size(); ++i) a[i].draw();
}

void f(Shape* q, vector<Circle>& s0)
{
    Polygon s1[10];
    Shape s2[20];
    // initialize
    Shape* p1 = new Rectangle(Point(0,0), Point(10,20));
    better(make_ref(s0)); // error: Array_ref<Shape> required
    better(make_ref(s1)); // error: Array_ref<Shape> required
    better(make_ref(s2)); // OK (no conversion required)
    better(make_ref(p1, 1)); // OK: one element
    delete p1;
    p1 = 0;
    better(make_ref(p1, 1)); // OK: no elements
    better(make_ref(q, max)); // OK (if max is OK)
}
```

新的程序有如下改进:

- 代码更简洁。程序员大多数情况下无需考虑大小, 即便某些时候需要考虑, 也仅仅局限于 `Array_ref` 初始化的部分, 而不会出现在代码其他位置。
- 解决了 `Circle[]` 转换为 `Shape[]`、`Polygon[]` 转换为 `Shape[]` 所存在的问题。
- 隐含地解决了 `s1`、`s2` 所存在的错误的元素数目问题。
- `max` 的潜在问题(以及其他的指针指向的元素数目问题)变得更为明显了, 这里是我们唯

—需要显式地处理大小的地方。

- 我们系统地、隐式地解决了空指针和空 vector 问题。

25.4.4 继承和容器

但是, 如果我们需要将 Circle 对象序列作为 Shape 对象序列来处理, 也就是说, 我们确实需要 better() (实际上是 draw_all() 的变形, 参见 19.3.2 节和 22.1.3 节) 来处理多态, 又该怎么办呢? 基本上, 这是办不到的。在 19.3.3 节和 25.4.2 节中, 我们已经看到, 类型系统有很充分的理由拒绝将 `vector<Circle>` 作为 `vector<Shape>` 来处理。基于同样理由, `Array_ref<Circle>` 也不能作为 `Array_ref<Shape>`。如果你忘记了这部分内容, 最好重新阅读 19.3.3 节, 因为这是一个非常基础的程序设计原则, 虽然它有些不方便。

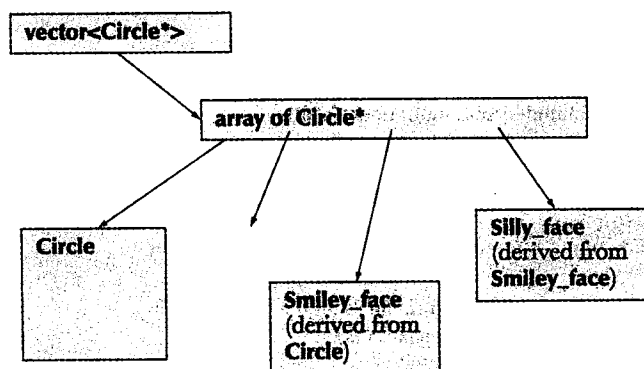
而且, 为了防止运行时的多态行为, 我们必须通过指针(或者引用)来访问多态对象: better() 中是不该使用 `p[i].draw()` 的。当我们一看到对多态对象使用点操作符而不是箭头 (`->`) 时, 就想到可能要出问题了。

那么我们应该怎么做呢? 首先, 必须使用指针(或引用)来访问对象, 因此, 在例子程序中应该使用 `Array_ref<Circle*>`、`Array_ref<Shape*>` 等, 而不是 `Array_ref<Circle>`、`Array_ref<Shape>` 等。

但是, 我们又不能将 `Array_ref<Circle*>` 转换为 `Array_ref<Shape*>`, 否则接下来的代码就可能将一些不是 Circle* 的元素放入 `Array_ref<Shape*>` 中。不过, 我们可以钻个空子:

- 在这个例子中, 我们并不想修改 `Array_ref<Shape*>`, 而只是想将形状画出来! 这是一个有趣而且有用的特例: 由于我们不修改 `Array_ref<Shape*>`, 上述不该将 `Array_ref<Circle*>` 转换为 `Array_ref<Shape*>` 的理由也就不成立了。
- 所有指针数组都具有相同的内存布局(不管指针指向的是什么类型的对象), 因此我们不会陷入 25.4.2 节所述的布局问题。

也就是说, 将 `Array_ref<Circle*>` 作为不可变的 (immutable) `Array_ref<Shape*>` 来处理, 不存在任何问题。接下来, 我们只要找到这样一种转换方法就可以了, 请看下图:



将这样一个 Circle* 数组作为一个不可变的 Shape* 来处理(利用 Array_ref), 在逻辑上是没有任何问题的。

看起来我们已经闯入专家领域了。事实上, 这个问题确实非常棘手, 用现有的工具是很难解决的。但是, 我们还是先来看一下如何为这个有问题但又很常见的接口模式(指针问题和元素数量问题, 参见 25.4.2 节)找到一种接近完美的解决方案吧。请记住: 不要为了显示自己的聪明而

进入“专家领域”。通常来说,最好的开发策略是从库中找到专家们已经设计实现好并已经过测试的工具,直接使用它们。

首先,我们重写 `better()`,对多态对象的访问全部改用指针,以确保我们不会“弄乱”给定的容器:

```
void better2(const Array_ref<Shape* const> a)
{
    for (int i = 0; i < a.size(); ++i)
        if (a[i])
            a[i]->draw();
}
```

由于改用了指针,所以我们必须检测指针是否为空。为了确保 `better2()` 不会通过 `Array_ref` 修改数组或向量的内容,我们使用了两个 `const`。第一个 `const` 保证我们不会对 `Array_ref` 使用修改(更新)操作,如 `assign()` 和 `reset()`。第二个 `const` 放在 `*` 之后,表示这是一个常量指针(不是指向常量内容的指针),即我们不希望修改指针本身(`Array_ref` 的元素)。

接下来,我们需要解决核心问题:如何表达如下意图?

- `Array_ref < Circle* >` 可以转换为类似 `Array_ref < Shape* >` 的东西(这样就能在 `better2()` 中使用)。
- 但是只能转换为不可变的 `Array_ref < Shape* >`。

我们可以通过定义一个转换运算符来实现上述目标:

```
template<class T>
class Array_ref {
public:
    // as before

    template<class Q>
    operator const Array_ref<const Q>()
    {
        // check implicit conversion of elements:
        static_cast<Q>(*static_cast<T*>(0));

        // cast Array_ref:
        return Array_ref<const Q>(reinterpret_cast<Q*>(p),sz);
    }

    // as before
};
```

这段代码有点令人头疼,不过基本要点如下:

- 类型转换运算符实现到 `Array_ref < const Q >` 的转换,对于给定的类型 `Q`,它先将 `Array_ref < T >` 的一个元素转换为 `Array_ref < Q >` 的元素(我们并不使用转换的结果,只是检验一下转换是否可行)。
- 接下来,转换运算符使用强制类型转换(`reinterpret_cast`)获得一个指定元素类型的指针,来构造新的 `Array_ref < const Q >`。强制转换通常会有额外开销,因此,不要对多重继承的类进行 `Array_ref` 类型转换(参见附录 A.12.4)。
- 请注意 `Array_ref < const Q >` 中的 `const`,它的作用就是保证不会将 `Array_ref < const Q >` 复制到老版本的可变的 `Array_ref < Q >` 中。

我们已经警告过你,你已经进入了“令人头疼”的“专家领域”。不过,这个版本的 `Array_ref` 还是比较容易使用的(令人头疼的只是定义和实现,而非应用):

```

void f(Shape* q, vector<Circle*>& s0)
{
    Polygon* s1[10];
    Shape* s2[20];
    // initialize
    Shape* p1 = new Rectangle(Point(0,0),10);
    better2(make_ref(s0));    // OK: converts to Array_ref<Shape*const>
    better2(make_ref(s1));    // OK: converts to Array_ref<Shape*const>
    better2(make_ref(s2));    // OK (no conversion needed)
    better2(make_ref(p1,1));   // error
    better2(make_ref(q,max)); // error
}

```

最后两条语句对指针的使用是错误的，因为两个指针是 `Shape*` 类型，而 `better2()` 需要的是一个 `Array_ref<Shape*` 类型的参数。也就是说，`better2()` 需要的是包含指针的容器，而非指针本身。如果我们希望将指针传递给 `better2()`，就必须将指针置于容器中（如内置数组或 `vector`）传递。对于一个单独的指针，我们可以使用 `make_ref(&p1, 1)`，虽然看起来有些笨拙，但能够达到目的。但是，对于数组（包含多于一个元素），如果不创建指向元素的指针，再置于一个容器中，是没有办法处理的。

总之，我们可以创建简单、安全、易于使用并且高效的接口，来弥补数组的不足。这就是本节的主要目的。“通过间接方式解决每个问题”（引自 David Wheeler）已经被作为“计算机科学第一定律”。这就是我们解决这个接口问题所采用的方法。

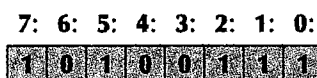
25.5 位、字节和字

在本书前面的章节中，我们已经讨论过内存硬件层次的一些概念，如位、字节和字。但在普通程序设计中，我们不会过多考虑这些概念，我们思考问题的方式是将数据看做特定类型的对象，如 `double`、`string`、`Matrix` 以及 `Simple_window`。在嵌入式程序设计中，我们必须对内存的低层组织方式有更多的了解，在本节中，我们会对此进行讨论。

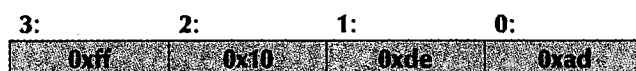
如果你对整数的二进制和十六进制表示的相关知识不太了解，请参考附录 A.2.1.1。

25.5.1 位和位运算

一个字节可以看做 8 个位的序列：



注意，位编号的习惯顺序是由右（最低有效位）至左（最高有效位）。类似地，一个字也可看做 4 个字节的序列：



编号顺序同样是由右至左，即从最低有效位到最高有效位。这两个图过分简化了现实世界中的情况：曾经存在一个字节有 9 位的计算机（虽然没有一台的寿命超过 10 年），而一个字包含两个字节的计算机就更常见了。不过，只要你记得在使用“8 位”和“4 字节”这两个特性之前查阅一下系统手册，就不会出现问题了。

如果希望程序是可移植的，那么请在程序中使用 `<limits>`（参见 24.2.1 节）以确保类型大小

不会弄错。

在 C++ 中我们如何来表示一组二进制位呢？答案取决于我们要处理多少位，以及希望哪些操作更方便和高效。我们可以将整型值当做一组二进制位来使用：

- bool——1 位，但占用整个字节的空间
- char——8 位
- short——16 位
- int——通常是 32 位，但在很多嵌入式系统中是 16 位
- long int——32 位或 64 位

上面列出的都是典型的类型大小，但在不同的实现中可能有所不同。因此，最稳妥的方法是实际测试一下。另外，标准库中也提供了处理位的方法：

- `std::vector<bool>`——当我们需要超过 $8 * \text{sizeof}(\text{long})$ 个二进制位时使用
- `std::bitset`——当需要超过 $8 * \text{sizeof}(\text{long})$ 个位时使用
- `std::set`——无序的、命名的二进制位集合（参见 21.6.5 节）
- 文件：海量的二进制位（参见 25.5.6 节）

而且，我们还可以使用如下两个语言特性来表示二进制位：

- 枚举(enum)，参见 9.5 节
- 位域，参见 25.5.5 节

这么多表示“位”的方法，从一个侧面反映了：在计算机内存中，实际上任何数据最终都表示为一组二进制位，因此人们迫切地需要提供很多方法来查看位、命名位以及完成位运算。注意，所有内置语言特性都是处理固定数量的二进制位（如 8、16、32 和 64），因此可以直接使用硬件提供的指令以最佳性能进行运算。与之相对，标准库特性都能处理任意数量的位。这可能会影响性能，但不要忙着下结论：如果你能将一组二进制位很好地映射到下层硬件，这些库特性通常都有很好的性能。

我们先来考察用整数表示二进制位的方式。C++ 提供了硬件直接支持的位运算，这些运算都是对运算对象逐位进行操作：

位运算		
	或	如果 x 的第 n 位为 1 或 y 的第 n 位为 1，则 x y 的第 n 位为 1
&	与	如果 x 的第 n 位为 1 且 y 的第 n 位为 1，则 x&y 的第 n 位为 1
^	异或	如果 x 的第 n 位为 1 或 y 的第 n 位为 1 且不同时为 1，则 x^y 的第 n 位为 1
<<	左移位	$x \ll s$ 的第 n 位是 x 的第 $n+s$ 位
>>	右移位	$x \gg s$ 的第 n 位是 x 的第 $n-s$ 位
~	补	$\sim x$ 的第 n 位是 x 的第 n 位的取反

你可能觉得将“异或”（ \wedge ，有时称为“xor”）作为一个基本运算有些奇怪，但在很多图形和加密程序中，异或是一个基本运算。

编译器不会把移位运算符“<<”误认为是一个输出操作符，但人有可能犯这样的错误。为了避免混淆，请记住输出操作符的左操作对象是一个 ostream，而移位运算符的左运算对象是一个整数。

注意，“&”与“&&”是不同的，“|”与“||”也是不同的，“&”和“|”会对运算对象的每一位独立进行计算（参见附录 A.5.5），计算结果的位数与运算对象相同。与之相反，“&&”和“||”只是

返回 true 或 false。

我们来尝试一些例子。我们常常用十六进制表示位的模式，下表列出了半字节值(4 位)的十六进制和二进制表示：

十六进制	位 模 式	十六进制	位 模 式
0x0	0000	0x8	1000
0x1	0001	0x9	1001
0x2	0010	0xa	1010
0x3	0011	0xb	1011
0x4	0100	0xc	1100
0x5	0101	0xd	1101
0x6	0110	0xe	1110
0x7	0111	0xf	1111

当数值小于 9 时，我们可以使用十进制，但使用十六进制可以提醒我们现在是在思考位模式。对于字节和字，十六进制非常有用。一个字节中的二进制位，可以表示为两个十六进制数字，例如：

十六进制字节	位模式
0x00	0000 0000
0x0f	0000 1111
0xf0	1111 0000
0xff	1111 1111
0xaa	1010 1010
0x55	0101 0101

在进行位运算时，使用 unsigned(参见 25.5.3 节)可以令情况更简单，避免一些不必要的问题。例如：

```
unsigned char a = 0xaa;
unsigned char x0 = ~a;    // complement of a

a:  1 0 1 0 1 0 1 0  0xaa
~a:  0 1 0 1 0 1 0 1  0x55

unsigned char b = 0x0f;
unsigned char x1 = a&b;    // a and b

a:  1 0 1 0 1 0 1 0  0xaa
b:  0 0 0 0 1 1 1 1  0xf
a&b: 0 0 0 0 1 0 1 0  0xa

unsigned char x2 = a^b;    // exclusive or: a xor b

a:  1 0 1 0 1 0 1 0  0xaa
b:  0 0 0 0 1 1 1 1  0xf
a^b: 1 0 1 0 0 1 0 1  0xa5
```

```
unsigned char x3 = a<<1; // left shift 1
```

a:  0xaa

a<<1:  0x54

注意, 最低位(第 0 位)填入了一个 0, 可以看做是从第 0 位右边“移来”的。而原来的最高位(第 7 位)被简单丢弃。

```
unsigned char x4 == a>>2; // right shift 2
```

a:  0xaa

a>>2:  0x2a

最高两位(第 6 位和第 7 位)都填入了 0, 可以看做是从第 7 位左边“移来”的, 最低两位(第 1 位和第 0 位)被简单丢弃。

在处理位运算时, 就可以像这样画出位模式, 这样的图示能使我们对比模式有一个很好的直观感觉。不过, 对于更复杂的例子, 手工画出位模式就太繁琐了。下面的这个小程序能将整数转换为二进制位描述形式:

```
int main()
{
    int i;
    while (cin>>i)
        cout << dec << i << "=="
            << hex << "0x" << i << "=="
            << bitset<8*sizeof(int)>(i) << "\n";
}
```

其中使用了标准库中的 bitset 来打印整数的某个位:

```
bitset<8*sizeof(int)>(i)
```

一个 bitset 是一组固定数量的二进制位。在本例中, 我们使用一个整数中所能容纳的那么多二进制位, 也就是 $8 * \text{sizeof}(\text{int})$, 并用整数 i 来初始化 bitset。

试一试 编译、运行这个例子程序, 试着输入一些整数, 体会二进制和十六进制表示形式。如果你对负数的表示形式感到迷惑, 请在阅读 25.5.3 节后再重试。

25.5.2 bitset

标准库模板类 bitset 是在 `<bitset>` 中定义的, 它用于描述和处理二进制位集合。每个 bitset 的大小是固定的, 在创建时指定:

```
bitset<4> flags;
bitset<128> dword_bits;
bitset<12345> lots;
```

默认情况下, bitset 被初始化为全 0, 但通常我们都会给它一个初始值, 可以是一个无符号的整数或者由 0 和 1 组成的字符串。例如:

```
bitset<4> flags = 0xb;
bitset<128> dword_bits(string("10101010101010"));
bitset<12345> lots;
```

这两段代码中, lots 被初始化为全 0, dword_bits 的前 112 位被初始化为全 0, 后 16 位由程序指定。如果你给出的初始化字符串中包含 0 和 1 之外的符号, bitset 会抛出一个 `std::invalid_argument` 异常:

```
string s;
cin>>s;
bitset<12345> my_bits(s); // may throw std::invalid_argument
```

常用的位运算符都可用于 bitset。例如，假定 b1、b2 和 b3 都是 bitset：

```
b1 = b2&b3; // and
b1 = b2|b3; // or
b1 = b2^b3; // xor
b1 = ~b2;   // complement
b1 = b2<<2; // shift left
b1 = b2>>3; // shift right
```

基本上，对于位运算而言，bitset 就像 unsigned int(参见 25.5.3 节)一样，只不过其大小任意，由用户指定。你能对 unsigned int 做什么(除了算术运算之外)，就能对 bitset 做什么。特别地，bitset 对 I/O 也很有用：

```
cin>>b;           // read a bitset from input
cout<<bitset<8>('c'); // output the bit pattern for the character 'c'
```

当读入 bitset 时，输入流会寻找 0 和 1。例如，如果输入下面内容：

```
10121
```

输入流会读入 101，21 会被留下。

对于字节和字，bitset 中的位是由右至左编号的(从最低有效位到最高有效位)。这样，第 7 位的值就是 2^7 ：

对于 bitset 而言，编号顺序不仅仅是遵循惯例的问题，还起到二进制位的索引下标的作用。例如：

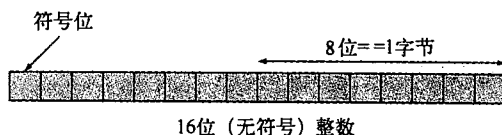
7:	6:	5:	4:	3:	2:	1:	0:
1	0	1	0	0	1	1	1

```
int main()
{
    const int max = 10;
    bitset<max> b;
    while (cin>>b) {
        cout<<b<<"\n";
        for (int i=0; i<max; ++i) cout<<b[i]; // reverse order
        cout<<"\n";
    }
}
```

如果你希望了解 bitset 的更多内容，请参考联机帮助、手册或者专业级的教材。

25.5.3 有符号数和无符号数

与大多数语言一样，C++ 同时支持有符号数和无符号数。无符号数在内存中的描述是很简单的：第 0 位表示 1、第 1 位表示 2，第 2 位表示 4，依此类推。但是，有符号数就引出一个问题：我们如何区分正数和负数？对此，C++ 给了硬件设计者一定的自由选择的余地，不过几乎所有实现都使用了二进制补码表示法。最靠左的二进制位(最高有效位)用来作为“符号位”：



如果符号位为 1，就表示负数。二进制补码表示法事实上已经成为标准方法。为了节约篇幅，我们只讨论如何在 4 位二进制整数中表示有符号数值：

正数:	0	1	2	4	7
	0000	0001	0010	0100	0111
负数:	1111	1110	1101	1011	1000
	-1	-2	-3	-5	-8

基本思想就是：用 x 的位模式的补码 ($\sim x$ ；参见 25.5.1 节) 来表示 $-(x+1)$ 的位模式。

到目前为止，我们一直在使用有符号整数（如 `int`）。更好的程序设计原则是：

- 当需要表示数值时，使用有符号数（如 `int`）。
- 当需要表示位集合时，使用无符号数（如 `unsigned int`）。

这是一个很好的程序设计原则，但很难严格遵循，因为一些人更喜欢用无符号数进行某些算术运算，而我们有时需要用这类代码。特别是还有一些历史遗留问题，例如，在 C 语言历史的早期，`int` 还是 16 位大小，每一位都很重要，而一个 `vector` 的大小 `v.size()` 返回的是一个无符号数。例如：

```
vector<int> v;
// ...
for (int i = 0; i < v.size(); ++i) cout << v[i] << '\n';
```

好的编译器会给出一个警告，指出存在有符号数（即 `i`）和无符号数（即 `v.size()`）混合运算的情况。有符号数和无符号数混合运算有可能会带来灾难性的后果。例如，循环变量 `i` 可能会溢出，即 `v.size()` 有可能比最大的有符号 `int` 值还要大。当 `i` 的值增大到有符号 `int` 所能表示的最大正数（2 的幂减 1，幂次等于 `int` 的二进制位数减 1，如 `int` 为 16 位宽度，此值为 $2^{15} - 1$ ）时，下一次增 1 运算不会得到更大的整数值，而会得到一个负数。因此循环永远也不会停止！每当我们到达最大整数时，接着就会从最小负 `int` 值重新开始。因此，如果 `v.size()` 的值为 $32 * 1024$ 或者更大，循环变量为 16 位 `int` 型的话，这个循环就是一个（可能非常严重的）bug。如果循环变量是 32 位 `int` 型的话，当 `v.size()` 的值大于等于 $2 * 1024 * 1024 * 1024$ 时就会出现同样的问题。

因此，严格来说，本书中的大部分循环都是有问题的。换句话说，对于嵌入式系统，我们要么证实循环不会达到临界点，要么将循环代码改写为另外一种形式。为了避免这个问题，我们可以使用 `vector` 提供的 `size_type` 或是迭代器：

```
for (vector<int>::size_type i = 0; i < v.size(); ++i) cout << v[i] << '\n';

for (vector<int>::iterator p = v.begin(); p != v.end(); ++p) cout << *p << '\n';
```

`size_type` 确保是无符号的，因此，第一种形式（使用无符号数）与 `int` 型循环变量的版本相比，多出一个二进制位来表示循环变量的数值（而不是符号）。这个改进很重要，但终究只是多出一位来表示循环的范围（循环次数多出一倍）。而使用迭代器的版本就不存在这个限制。

试一试 下面这个例子看起来没什么问题，但它实际上是个死循环：

```
void infinite()
{
    unsigned char max = 160;    // very large
    for (signed char i=0; i<max; ++i) cout << int(i) << '\n';
}
```

运行这个程序，解释为什么会形成死循环。

基本上，我们将无符号数当做整数来使用有两个原因，而不是简单作为一组二进制位（即不使用 `+`、`-`、`*` 和 `/`）：

- 有更多的二进制位来表示数值，从而获得更高的精度。

- 用来表示逻辑属性，其值不能是负数。

前者就是我们刚刚看到的，使用无符号循环变量带来的效果。

混合使用有符号数和无符号数的问题在于，在 C++ 中（在 C 中也一样），两者转换的方式很奇怪，而且难以记忆。例如：

```
unsigned int ui = -1;
int si = ui;
int si2 = ui+2;
unsigned ui2 = ui+2;
```

奇怪的是，第一个初始化操作能够成功完成，ui 被赋予 4294967295，这个 32 位无符号整数的位模式恰好与有符号数 -1 相同（二进制表示为“全 1”）。一些人认为这样编写代码很简洁，因此喜欢用 -1 表示“全 1”，而另外一些人则认为这是一个不好的特性。从无符号数到有符号数的转换规则与此类似，因此第二个初始化操作将 si 的初值设置为 -1。如我们所料，si2 被赋予 1（-1 + 2 == 1），ui2 也被赋予同样的值。ui2 的计算结果应该会让你感到惊讶：为什么 4294967295 + 2 会得到 1？如果我们将 4294967295 表示为十六进制 0xffffffff，就比较清楚了：4294967295 是最大的 32 位无符号整数，因此 4294967297 无法用 32 位整数表示，无论是无符号或是有符号都不行。我们可以说 4294967295 + 2 产生了溢出，或者更准确地说，这里使用了模运算。也就是说，32 位整数运算都要对 2^{32} 取模。

现在所有事情都清楚了吗？即使你已经弄清了这些奇怪的规则，我们也希望上述讨论能使你信服这样一个观点：用无符号数表示数值，以获得一个额外的二进制位的精度，无异于玩火，这样做会导致混乱，而且是潜在的错误之源。

如果发生整数溢出，会有什么后果呢？考虑下面代码：

```
int i = 0;
while (++i) print(i); // print i as an integer followed by a space
```

这段程序会输出什么样的数值序列呢？显然，这取决于 Int 是如何定义的（注意，这里大写的 I 并不是打字错误）。对任何一种大小有限制的整数类型，最终都会出现溢出的情况。如果 Int 是无符号类型（如 unsigned char、unsigned int 或 unsigned long long），由于“++”运算会进行模运算，循环变量 i 达到最大值后会变为 0（循环从而终止）。如果 Int 是有符号类型（如 signed char），i 达到最大值后会突然变为最小的负数然后逐渐增大为 0（循环终止）。例如，如果 Int 是 signed char，输出的序列为 1 2...126 127 -128 -127...-2 -1。

再次提出那个问题：如果发生整数溢出会有什么后果？答案是程序还会继续执行，就好像有更多二进制位保存结果一样，但实际上一些无法容纳的二进制被丢弃了。一般的策略是丢弃最靠左的位（最高有效位）。如果在赋值语句中赋予变量一个超出其表示范围的值，也会看到类似的效果：

```
int si = 257; // doesn't fit into a char
char c = si; // implicit conversion to char
unsigned char uc = si;
signed char sc = si;
print(si); print(c); print(uc); print(sc); cout << '\n';

si = 129; // doesn't fit into a signed char
c = si;
uc = si;
sc = si;
print(si); print(c); print(uc); print(sc);
```

输出结果为

```
257    1      1      1
129   -127   129   -127
```

产生这样的结果的原因是, 257 比 8 个二进制位所能表示的最大值(255, 即“8 个 1”)大 2; 129 比 7 个二进制位所能表示的最大值(127, 即“7 个 1”)大 2, 因而符号位被置位, 有符号变量的值变为负数。注意, 程序的运行结果表明: 在我们的计算机上, char 是无符号的, 因为 c 的行为与 uc 一致, 而与 sc 不同。

试一试 在纸上画出上面程序中涉及的位模式, 算出若 si = 128, 输出结果是什么。

运行程序, 检验你的手算结果是否正确。

插一句: 我们为什么要引入 print() 函数? 我们可以试试:

```
cout << i << ' ';
```

原因很简单, 如果 i 是 char 型, 这条语句就会输出一个字符, 而不是其整数值。因此, 我们引入 print() 函数, 对所有整数类型进行一致处理, 定义如下:

```
template<class T> void print(T i) { cout << i << '\t'; }
```

```
void print(char i) { cout << int(i) << '\t'; }
```

```
void print(signed char i) { cout << int(i) << '\t'; }
```

```
void print(unsigned char i) { cout << int(i) << '\t'; }
```

总结一下: 无符号数的使用可以和有符号数完全一样(包括普通的算术运算), 但是要避免这样使用, 因为这样做会使问题变得非常复杂, 程序也容易出错。

- 永远不要为了多出一个二进制位的精度而用无符号数表示数值。
- 如果你需要一个额外的二进制位, 很快就会再需要一个。

不幸的是, 无符号数的算术运算是很难完全避免的:

- 标准库容器的下标都是无符号数。
- 一些人就是喜欢无符号数的算术运算。

25.5.4 位运算

我们为什么需要位运算呢? 好吧, 实际上大多数人并不喜欢位运算。位处理靠近下层而且容易出错, 有可能的话就应该尽量使用替代方法。但是, 位描述和位运算是非常基础的, 也是非常有用的, 我们不能假装它不存在。这听起来有些消极, 有些令人气馁, 但值得仔细思考。一些人确实喜欢摆弄位和字节, 因此应当记住: 位处理在某些时候是不可避免的(虽然开始时有些不情愿, 但在过程中你很可能获得不少乐趣), 但不要在程序中到处使用位运算。John Bentley 的两句话用在这里很适合: “喜弄位者易被位反噬(bitten)”, “喜弄字节者易被字节反噬(bytten)”。

那么, 什么时候应该使用位运算呢? 有些情况下, 应用程序要处理的对象就是位的形式, 那么使用位运算就是顺理成章的了。这方面的例子包括硬件指示器(“标识位”)、低层通信(需要从字节流中提取不同类型的值)、图形应用(需要用多个层次的图像组成图片)以及加密(参见下一节)。

例如, 考虑如何从一个整数中提取(低层)信息(可能我们想将它按字节传输, 就像二进制 I/O 的处理方式):

```

void f(short val)    // assume 16-bit, 2-byte short integer
{
    unsigned char left = val&0xff;           // leftmost (least significant) byte
    unsigned char right = (val>>8)&0xff;     // rightmost (most significant) byte
    // ...
    bool negative = val&0x8000;             // sign bit
    // ...
}

```

这种运算是很常见的，通常称为“移位和掩码”运算。“移位”运算(使用“<<”或“>>”)将二进制位移动到我们所期望的位置(本例中是移动到字的最低有效位)，以方便处理。“掩码”运算是将运算对象与一个特殊的位模式(本例中是 0xff)进行“位与”(&)运算，目的是去掉那些我们不需要的位。

如果希望对二进制位命名，我们通常使用枚举类型，例如：

```

enum Printer_flags {
    acknowledge=1,
    paper_empty=1<<1,
    busy=1<<2,
    out_of_black=1<<3,
    out_of_color=1<<4,
    // ...
};

```

每个枚举常量被赋予的值与名字的含义是完全吻合的：

out_of_color	16	0x10	0001 0000
out_of_black	8	0x8	0000 1000
busy	4	0x4	0000 0100
paper_empty	2	0x2	0000 0010
acknowledge	1	0x1	0000 0001

这种常量值在某些情况下是很有用的，因为它们可以任意组合：

```

unsigned char x = out_of_color | out_of_black; // x becomes 24 (16+8)
x |= paper_empty;                             // x becomes 26 (24+2)

```

注意，这里的“|”起到了“置位”的作用。类似地，“&”可以起到“检测位”的作用，例如：

```

if (x& out_of_color) { // is out_of_color set? (yes, it is)
    // ...
}

```

命名二进制位同样可以用来进行掩码运算：

```

unsigned char y = x&(out_of_color | out_of_black); // x becomes 24

```

此时，y 的内容就是 x 的第 3 位和第 4 位(out_of_black 和 out_of_color 对应第 3、4 位)。

将 enum 作为二进制位集合来使用，是一种十分常用的方法。此时，我们就需要一种方法将位运算的计算结果再“转换回”enum：

```

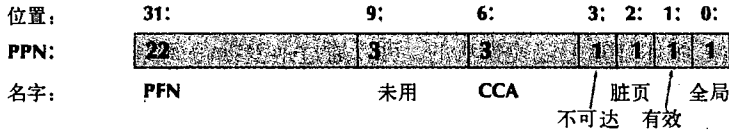
Flags z = Printer_flags(out_of_color | out_of_black); // the cast is necessary

```

之所以需要这样的类型转换，是因为编译器不知道 out_of_color | out_of_black 的值是否是合法的 Flags 值。编译器的怀疑是合理的：毕竟，没有任何一个枚举常量的值为 24(out_of_color | out_of_black 的计算结果)。当然，在本例中，我们知道赋值语句是合理的(但编译器并不知道)。

25.5.5 位域

如前所述，硬件接口是使用位运算最多的地方。通常，接口就是一组二进制位和不同大小的数。“二进制位和数”通常是命名的，出现在字的不同位置，我们称之为“设备寄存器”。C++ 提供了一种特殊的语言特性来处理这种固定的数据布局：位域(bitfields)。我们来考察这样一个例子：操作系统中页管理程序所使用的页号，其结构为：



可以看到, 一个 32 位的字被分为两个数值域(一个占用 22 位, 一个占用 3 位)和 4 个标识位, 这些数据的大小和位置是固定的。在字的中间还有一个“未用”域。此数据布局可用如下 struct 类型来描述:

```
struct PPN { // R6000 Physical Page Number
    unsigned int PFN : 22; // Page Frame Number
    int : 3; // unused
    unsigned int CCA : 3; // Cache Coherency Algorithm
    bool nonreachable : 1;
    bool dirty : 1;
    bool valid : 1;
    bool global : 1;
};
```

我们必须查阅参考手册才知道 PFN 和 CCA 应该定义为无符号整数, 但如果没有手册的帮助, 我们可以直接利用位模式图来设计 struct。位域在字中是由左至右排列的。每个域的位宽用一个整数指定, 名字和位宽之间用冒号隔开。不允许为位域指定绝对的位置(如第 8 位)。如果总位宽超出了一个字的容纳能力, 则超出部分被置于下一个字中。希望这种方式能满足你的需求。定义完成后, 位域的使用就与其他变量没什么差别了:

```
void part_of_VM_system(PPN * p)
{
    // ...
    if (p->dirty) { // contents changed
        // copy to disk
        p->dirty = 0;
    }
    // ...
}
```

如果没有位域, 要想获得一个字中间区域的信息, 就必须使用复杂的移位和掩码运算, 位域使这一切变得简单。例如, 对于一个 PPN 类型的对象 pn, 可以这样提取 CCA:

```
unsigned int x = pn.CCA; // extract CCA
```

如果是用一个 int 型变量 pni 表示页号, 则必须这样来提取 CCA:

```
unsigned int y = (pni >> 4) & 0x7; // extract CCA
```

也就是说, 先将 CCA 右移到最低有效位, 然后与 0x7 (即最右 3 位置位) 进行掩码运算来去掉所有其他位。你可以查看一下编译得到的机器码, 多半会发现生成的代码就是这两个指令。

CCA、PPN 和 PFN 这种字头缩写形式是常见的低层程序设计风格, 显然, 在设计普通程序时, 这并不是一种好的风格。

25.5.6 实例: 简单加密

接下来, 我们实现一个简单的加密算法: 微型加密算法 (Tiny Encryption Algorithm, TEA), 作为位/字节级数据处理的一个实例。这个算法最初是由剑桥大学的 David Wheeler 设计的 (参见 22.2.1 节)。它很简单, 但应付一般攻击还是绰绰有余的。

你不必过于仔细地阅读加密程序 (除非你真的需要理解算法, 而且对困难有心理准备)。我们给出这个加密程序只是为了让你体会一下如何编写实用的位处理代码。如果你希望学习加密的知识, 请查阅专门的教材。至于用其他语言实现 TEA 算法的相关内容, 请参考 http://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm 以及英国布拉福德大学 Simon Shepherd 教授关于 TEA 的网站。

加密/解密的基本思想是很简单的。我想发送给你一些文本, 但我不想让其他人看懂发送的

内容。因此，我先把要发送的文本进行转换，然后再发送，使得不知道确切转换方式的人就无法看懂转换后的内容；而你是知道转换方法的，可以通过逆变换得到原始文本。这个转换过程就称为加密。进行加密需要一个算法（我们必须假定所有人都能获得这个算法）和一个称为“密钥”的字符串。你和我都知道密钥（我们希望窃听者不知道）。当你获得密文时，可以使用“密钥”对其进行解密，即重新构造出我要发送的“明文”。

TEA 算法接受三个参数， v 是包含两个无符号 $\text{long}(v[0], v[1])$ 的数组，表示要加密的 8 个字符， w 是用来保存密文的，也是包含两个无符号 $\text{long}(w[0], w[1])$ 的数组，而 k 是密钥，是包含 4 个无符号 $\text{long}(k[0] \cdots k[3])$ 的数组：

```
void encipher(
    const unsigned long *const v,
    unsigned long *const w,
    const unsigned long *const k)
{
    unsigned long y = v[0];
    unsigned long z = v[1];
    unsigned long sum = 0;
    unsigned long delta = 0x9E3779B9;
    unsigned long n = 32;
    while(n-- > 0) {
        y += (z << 4 ^ z >> 5) + z ^ sum + k[sum&3];
        sum += delta;
        z += (y << 4 ^ y >> 5) + y ^ sum + k[sum>>11 & 3];
    }
    w[0]=y; w[1]=z;
}
```

注意，所有数组都是无符号类型，这样我们就可以放心地进行位运算，而不必担心突然出现负数。移位（<<和>>）、异或（^）以及位与（&）这些位运算结合无符号数加法运算形成了完整的加密算法。这段代码只能用于 long 的大小是 4 字节的机器。也就是说，代码中散布着“魔数”（即 $\text{sizeof}(\text{long}) == 4$ ）。如前所述，这通常不是一种好的程序设计策略，但这样写程序，代码能放在一页纸内。而相应的数学公式也能写在一个信封的背面，或者按照最初的设想，牢牢地记在程序员的头脑中。David Wheeler 最初设计算法时，就希望加密算法如此简单：在他外出旅行忘带笔记和便携式电脑的情况下，仅凭头脑也能回忆起算法，完成加密操作。除了简洁，这段代码还很快。变量 n 的值决定了循环次数：循环次数越多，加密强度越高。据我们所知，当 $n == 32$ 时，TEA 尚未被攻破过。

下面是解密函数：

```
void decipher(
    const unsigned long *const v,
    unsigned long *const w,
    const unsigned long *const k)
{
    unsigned long y = v[0];
    unsigned long z = v[1];
    unsigned long sum = 0xC6EF3720;
    unsigned long delta = 0x9E3779B9;
    unsigned long n = 32;
    // sum = delta<<5, in general sum = delta * n
    while(n-- > 0) {
        z -= (y << 4 ^ y >> 5) + y ^ sum + k[sum>>11 & 3];
        sum -= delta;
        y -= (z << 4 ^ z >> 5) + z ^ sum + k[sum&3];
    }
    w[0]=y; w[1]=z;
}
```

如果文件需要在不安全的信道上传输，我们可以像下面代码这样对其加密：

```
int main()    // sender
{
    const int nchar = 2*sizeof(long);    // 64 bits
    const int kchar = 2*nchar;          // 128 bits

    string op;
    string key;
    string infile;
    string outfile;
    cout << "please enter input file name, output file name, and key:\n";
    cin >> infile >> outfile >> key;
    while (key.size()<kchar) key += '0';    // pad key
    ifstream inf(infile.c_str());
    ofstream outf(outfile.c_str());
    if (!inf || !outf) error("bad file name");

    const unsigned long* k =
        reinterpret_cast<const unsigned long*>(key.data());

    unsigned long outptr[2];
    char inbuf[nchar];
    unsigned long* inptr = reinterpret_cast<unsigned long*>(inbuf);
    int count = 0;

    while (inf.get(inbuf[count])) {
        outf << hex;    // use hexadecimal output
        if (++count == nchar) {
            encipher(inptr, outptr, k);
            // pad with leading zeros:
            outf << setw(8) << setfill('0') << outptr[0] << ' '
                << setw(8) << setfill('0') << outptr[1] << ' ';
            count = 0;
        }
    }

    if (count) {    // pad
        while(count != nchar) inbuf[count++] = '0';
        encipher(inptr, outptr, k);
        outf << outptr[0] << ' ' << outptr[1] << ' ';
    }
}
```

程序最重要的部分是 while 循环，剩余部分只是起辅助作用。while 循环读入字符存到输入缓冲区 inbuf 中，每次都 8 个字符传递给 encipher() 进行加密。TEA 不关心传递给它的字符，实际上它并不知道自己加密的是什么。例如，你可能在加密一幅照片或者一次电话通话。TEA 所关心的只是接受 64 位（两个无符号 long）明文，生成 64 位密文。因此，我们用一个指针指向 inbuf，将其转换为 unsigned long* 类型并传递给 TEA。对密钥也是相同的处理方式。由于 TEA 使用 128 位的密钥（4 个 unsigned long），因此我们对用户输入“打补丁”，将其补齐为 128 位。最后一条语句将 0 补在文本末尾，使其位数变为 TEA 所要求的 64 的整数倍（8 个字节）。

密文如何传输呢？我们可以自由选择传输方法，但要注意的是，由于密文是二进制位序列，而不是 ASCII 或 Unicode 字符，因此不能像普通文本一样传输。可以选择二进制 I/O 方法（参见 11.3.2 节），不过在本例中我们用十六进制数的形式输出密文。

```

5b8fb57c 806fbcce 2db72335 23989d1d 991206bc 0363a308
8f8111ac 38f3f2f3 9110a4bb c5e1389f 64d7efe8 ba133559
4cc00fa0 6f77e537 bde7925f f87045f0 472bad6e dd228bc3
a5686903 51cc9a61 fc19144e d3bcde62 4fdb7dc8 43d565e5
f1d3f026 b2887412 97580690 d2ea4f8b 2d8fb3b7 936cfa6d
6a13ef90 fd036721 b80035e1 7467d8d8 d32bb67e 29923fde
197d4cd6 76874951 418e8a43 e9644c2a eb10e848 ba67dcd8
7115211f db32069 e4e92f87 8bf3e33e b18f942c c965b87a
44489114 18d4f2bc 256da1bf c57b1788 9113c372 12662c23
eeb63c45 82499657 a8265f44 7c866aae 7c80a631 e91475e1
5991ab8b 6aedbb73 71b642c4 8d78f68b d602bfe4 d1eadde7
55f20835 1a6d3a4b 202c36b8 66a1e0f2 771993f3 11d1d0ab
74a8cfda 4ce54f5a e5fda09d acbdf110 259a1a19 b964a3a9
456fd8a3 1e78591b 07c8f5a2 101641ec d0c9d7e1 60dbeb11
b9ad8e72 ad30b839 201fc553 a34a79c4 217ca84d 30f666c6
d018e61c d1c94ea6 6ca73314 cd60def1 6e16870e 45b94dc0
d7b44fcd 96e0425a 72839f71 d5b6427c 214340f9 8745882f
0602c1a2 b437c759 ca0e3903 bd4d8460 edd0551e 31d34dd3
c3f943ed d2cae477 4d9d0b61 f647c377 0d9d303a ce1de974
f9449784 df460350 5d42b06c d4dedb54 17811b5f 4f723692
14d67edb 11da5447 67bc059a 4600f047 63e439e3 2e9d15f7
4f21bbbe 3d7c5e9b 433564f5 c3ff2597 3a1ealdf 305e2713
9421d209 2b52384f f78fbae7 d03c1f58 6832680a 207609f3
9f2c5a59 ee31f147 2ebc3651 e017d9d6 d6d60ce2 2be1f2f9
eb9de5a8 95657e30 cad37fda 7bce06f4 457daf44 eb257206
418c24a5 de687477 5c1b3155 f744fbff 26800820 92224e9d
43c03a51 d168f2d1 624c54fe 73c99473 1bce8fbb 62452495
5de382c1 1a789445 aa00178a 3e583446 dcdbd64c5 ddda1e73
fa168da2 60bc109e 7102ce40 9fed3a0b 44245e5d f612ed4c
b5c161f8 97ff2fc0 1dbf5674 45965600 b04c0afa b537a770
9ab9bee7 1624516c 0d3e556b 6de6eda7 d159b10e 71d5c1a6
b8bb87de 316a0fc9 62c01a3d 0a24a51f 86365842 52dabf4d
372ac18b 9a5df281 35c9f8d7 07c8f9b4 36b6d9a5 a08ae934
239efba5 5fe3fa6f 659df805 faf4c378 4c2048d6 e8bf4939
31167a93 43d17818 998ba244 55dba8ee 799e07e7 43d26aef
d5682864 05e641dc b5948ec8 03457e3f 80c934fe cc5ad4f9
0dc16bb2 a50aa1ef d62ef1cd f8fbbf67 30c17f12 718f4d9a
43295fed 561de2a0

```

试一试 如果密钥是 bs, 明文是什么?

这个程序还不是很完美, 任何安全专家都会告诉你, 将明文和密文保存在一起是个笨主意, 对于打补丁、密钥长度为 2 等问题也会提出看法。不过, 本书是一本程序设计书籍, 而非计算机安全书籍。

我们测试这个程序的方法是: 读入密文, 进行解密, 与明文比较。当编写程序时, 能进行简单的正确性测试总是好的。

下面是解密程序的核心部分:

```

unsigned long inptr[2];
char outbuf[nchar+1];
outbuf[nchar]=0; // terminator
unsigned long* outptr = reinterpret_cast<unsigned long*>(outbuf);
inf.setf(ios_base::hex, ios_base::basefield); // use hexadecimal input

while (inf>>inptr[0]>>inptr[1]) {
    decipher(inptr, outptr, k);
    outf<<outbuf;
}

```

注意这条语句:

```
inf.setf(ios_base::hex,ios_base::basefield);
```

它读入十六进制数。对于解密程序而言,这些数据保存在输出缓冲区 outbuf 中,进行类型转换后作为二进制位进行处理。

TEA 这个例子属于嵌入式系统程序设计范畴吗?这不太明确,但你可以想象它被用于安全系统或金融业务系统,这些应用都是由很多“小设备”组成的。无论如何,TEA 程序展示了很多好的嵌入式程序设计方法:它基于一个清晰的(数学)模型,能确保其正确性,它很简洁、很快速而且直接依赖于硬件特性。encipher() 和 decipher() 的接口风格并不很符合我们的习惯。但是,它们不仅用 C++ 实现,还用 C 实现,因此不能使用 C 语言不支持的 C++ 特性。另外,程序中出现的很多“魔数”都直接来自于数学模型。

25.6 编码规范

错误的源头是多种多样的。最严重、最难以修正的错误都是与上层设计决策相关的,这些设计决策包括总体的错误处理策略、遵循(或不遵循)特定的标准、算法设计、数据表示方式等。这些问题已经超出了本书的讨论范围,我们讨论的重点是那些由于糟糕的程序设计方式而导致的错误。“糟糕的程序设计方式”主要是指使用语言特性的方式容易引起错误,或者表达思想的方式模糊不清。

编码规范试图解决后一类问题,它定义一个“排版风格”来为程序员指引方向:对于特定应用,使用哪些 C++ 语言特性比较恰当。例如,嵌入式程序设计编码规范可能会禁止使用 new。通常,编码规范还会尽力令不同程序员编写出的代码更为相似,虽然他们可以自由选择程序设计风格。例如,某个编码规范可能会要求循环结构都用 for 语句实现(即禁止 while 语句)。这能使代码更一致,在开发大型项目时,这对代码维护有着重要作用。请注意,一种编码规范只针对一类特定的程序设计,只为某些特定的程序员提供帮助。不存在一种适合于所有 C++ 应用和所有 C++ 程序员的编码规范。

编码规范试图解决的是解决方案表达方式方面的问题,而不是应用的复杂性方面的问题。因此,我们可以说,编码规范试图解决偶然复杂性,而不是必然复杂性。

引起偶然复杂性的主要原因包括:

- 过于聪明的程序员,他们在表达复杂解决方案时试图使用那些并不理解或并不喜欢的语言特性。
- 未经良好培训的程序员,不会使用最适合的语言特性和库功能。
- 不必要的程序设计风格变化,这会导致完成相似工作的代码在形式上差异很大,给代码维护人员带来困扰。
- 不恰当的程序设计语言,这会导致所使用的语言特性非常不适合于某些应用领域或某些程序员。
- 没有有效利用库,导致程序中存在大量专门处理低层资源的代码。
- 不恰当的编码规范,导致解决某些问题时,付出额外的工作量或者无法采用最优的解决方案,从而引起一些棘手的问题。

25.6.1 编码规范应该是怎样的

一个好的编码规范应该能帮助程序员写出好的代码,即对于一些小的程序设计问题能够直接给出答案,而无需程序员花费时间逐个解决。有一句在工程师间流传很久的格言:“形式即解

放”。理想情况下，编码规范应该是指示性的，指出应该做什么。看起来显然应该这样，但很多编码规范只是简单罗列了不能做什么，而没有指明应该做什么。仅仅告诉程序员什么不能做不会对编程有什么帮助，而且常常会令人很恼火。

好的编码规范中的原则应该都是可验证的，最好可以通过程序来验证。也就是说，当我们写完程序后，查看一下代码就可以很容易地回答这个问题：“我是否违反了编码原则？”

对于列出的编码原则，一个好的编码规范应该解释清楚其理论依据。不能只是对程序员说：“我们就是这样做的！”，这只会增加程序员的厌恶感。更糟的是，如果程序员觉得规范的某些部分毫无益处，甚至妨碍他们写出高质量的程序，就会不停地尝试推翻它。不要期待编码规范的全部内容都受到欢迎。即使是最好的编码规范也都是一定程度上的折衷，而且大多数“不该做”都会引起问题，即便你自己没碰到。例如，不一致的命名规范是混乱之源，但不同人都有自己特别偏好的命名规范，而强烈抵触其他规范。例如，我个人认为首字母大写的标识符命名法（如 Camel-CodingStyle）“非常丑陋”，强烈倾向于“下划线风格”（如 `underscore_style`），认为这种风格更清晰、本质上更易读，很多人也赞同我的观点。但另一方面，也有很多人并不赞同这一观点。显然，没有任何一种命名规范能满足所有人，但多数情况下，一个一致风格绝对比没有规范更好。

关于编码规范应该是怎样的，我们总结如下：

- 一个好的编码规范应该针对特定应用领域和特定程序员设计。
- 一个好的编码规范应该既有指示性，又有限制性。
 - 推荐一些“基础的”库功能作为指示性原则，通常是最有效的方式。
- 一个编码规范就是一个编码原则集合，指明了程序风格。
 - 通常应该指定命名和缩进原则：如“使用‘Stroustrup 布局风格’”。
 - 通常应该指定允许使用的语言子集：如“不要使用 `new` 或 `throw`”。
 - 通常应该指定注释原则：如“每个函数应该用一段注释描述其功能”。
 - 通常应该指明使用哪些库：如“使用 `<iostream>` 而不是 `<stdio.h>`”或“使用 `vector` 和 `string` 而不是内置数组和 C 风格字符串”。
- 大多数编码规范的目标是提高程序的
 - 可靠性
 - 可移植性
 - 可维护性
 - 可测试性
 - 重用性
 - 可扩展性
 - 可读性
- 一个好的编码规范要比没有规范更好。如果没有编码规范，就不应该启动一个大型（需要很多人，多年才能完成）的工业项目。
- 一个糟糕的编码规范甚至比没有规范更糟。例如，一个限制使用 C 语言子集的 C++ 编程规范是有害的。但不幸的是，糟糕的编码规范并不罕见。
- 任何一个编码规范，即便是好的编码规范，也都会有程序员不喜欢。大多数程序员希望按自己喜欢的方式编写代码。

25.6.2 编码原则实例

下面，我们会给你列出一些编码规范中的编码原则。自然，我们之所以选择这些原则，就是

希望它们能对你有所帮助。但是,我们所见过的实际的编码规范还没有少于 35 页的,大多数还要长得多。所以,在本节中我们不会给出一个完整的编码规范。而且,如前所述,任何好的编码规范都是为特定应用领域和特定程序员所设计的。因此,我们不会伪称这些编码原则是通用的。

我们为编码原则编了号,并为每条原则给出了简短的设计依据。为了帮助理解,很多原则都附带了一些例子。我们将编码原则分为推荐规则(recommendation)和严格规则(firm rule)两类,对于前者,程序员偶尔可以不遵守,而后者则必须严格遵守。对于现实中的编码规范,只有管理者才能授权修改严格规则。如果你在程序中违反了推荐规则或者严格规则,应该通过注释来说明原因。在规范中,原则的例外情况也可与原则一并列出。本节中给出的每条严格原则都用一个大写字母 R 及其编号标识,而推荐原则都用小写字母 r 及其编号标识。

我们将编码原则分类如下:

- 一般原则
- 预处理原则
- 命名和布局原则
- 类原则
- 函数和表达式原则
- 硬实时原则
- 关键系统原则

“硬实时”和“关键系统”原则仅用于硬实时和关键系统程序设计。

与一个好的实际编码规范相比,我们使用的有些术语并不明确(如“关键”的确切含义是什么),而列出的原则也有些过于简单。你会发现它们与 JSF++ 原则(参见 25.6.3 节)有相似之处,这并不是偶然的,我本人参与了 JSF++ 原则的规划。不过,本书中的代码实例并未遵循本节中给出的编码原则,毕竟,本书中的程序并不是为关键的嵌入式系统所编写的。

一般原则

R100: 任何函数和类的代码规模都不应超过 200 行(不包括注释)。

原因: 长的函数和类会更复杂,因而难于理解和测试。

r101: 任何函数和类都应该能完全显示在一屏上,并完成单一的逻辑功能。

原因: 如果程序员只能看到函数或类的一部分,就很可能漏掉有错误的部分。如果一个函数试图完成多个功能,与单功能的函数相比,其规模就可能很大,而且会更复杂。

R102: 所有代码都应该遵循 ISO/IEC 14882: 2003(E) C++ 标准。

原因: 在 ISO/IEC 14882 标准之上的扩展和变形可能会不稳定,定义不明确,而且可能影响可移植性。

预处理原则

R200: 除了用于源码控制的 `#ifdef` 和 `#ifndef` 之外,不要使用宏。

原因: 宏不遵守定义域和类型规则,而且使代码变得更不清晰、不易读。

R201: `#include` 只能用于包含头文件(`*.h`)。

原因: `#include` 用于访问接口的声明而非实现细节。

R202: 所有 `#include` 语句都应位于任何非预处理声明之前。

原因: 如果 `#include` 语句位于程序中间,就很可能被阅读程序的人忽略,而且容易导致程序不同部分对名字的解析不一致。

R203: 头文件(*.h)不应包含非常量变量的定义或非内联、非模板函数定义。

原因: 头文件应该包含接口声明而非实现细节。但是, 常量通常被看做接口的一部分; 出于性能的考虑, 一些非常简单的函数应该作为内联函数(因此应该放在头文件中); 而当前的编译器要求完整的模板定义都放在头文件中。

命名和布局原则

R300: 应该使用缩进, 并且在一个源码文件中缩进风格应该一致。

原因: 可读性和代码风格。

R301: 每条新语句都另起一行。

原因: 可读性。

例子:

```
int a = 7; x = a+7; f(x,9);    // violation
int a = 7;    // OK
x = a+7;    // OK
f(x,9);    // OK
```

例子:

```
if (p<q) cout << *p;    // violation
```

例子:

```
if (p<q)
    cout << *p;    // OK
```

R302: 标识符的名字应该都具有描述性。

标识符可以包含常见的缩写和字头缩略。

如果 x、y、i、j 等是按习惯方式使用, 可以认为是有描述性的。

使用下划线风格(number_of_elements)而不是字头缩略风格(numberOfElements)。

不要用匈牙利命名法。

类型、模板和名字空间的命名都以大写字母开头。

避免过长的名字。

例子: Device_driver 和 Buffer_pool。

原因: 可读性。

注意: C++ 标准规定, 以下划线开头的标识符留作语言实现所用, 因此在用户程序中应被禁止。

例外: 调用经过认证的库, 来自库中的名字是可以使用的。

例外: 宏名用于保护#include 不被重复包含。

R303: 标识符不能只在以下方面不同:

- 大小写不同
- 只相差下划线
- 只是字母 O、数字 0 或字母 D 间的替换
- 只是字母 I、数字 1 或字母 l 之间的替换
- 只是字母 S 和数字 5 之间的替换
- 只是字母 Z 和数字 2 之间的替换
- 只是字母 n 和字母 h 之间的替换

例子: Head 和 head // 违反了原则。

原因：可读性。

R304：标识符不能只包含大写字母和下划线。

例子：BLUE 和 BLUE_CHEESE // 违反了原则。

原因：全部大写字母的标识符被广泛用于宏名，可能用于经过认证的库中的#include 文件，而不应该用于用户程序。

函数和表达式原则

r400：内层循环的标识符和外层循环的标识符不应重名。

原因：可读性和代码风格。

例子：

```
int var = 9; { int var = 7; ++var; } // violation: var hides var
```

R401：声明的作用域应该尽量小。

原因：保持变量的初始化和使用尽量靠近，以降低混乱的可能性；令离开作用域的变量释放其资源。

R402：所有变量都要初始化。

例子：

```
int var; // violation: var is not initialized
```

原因：未初始化的变量通常是错误之源。

例外：如果数组或容器会立即从输入接收数据，则不必初始化。

R403：不应使用类型转换。

原因：类型转换是错误之源。

例外：dynamic_cast 可以使用。

例外：新风格的类型转换可以使用，用来将硬件地址转换为指针，或者将从程序外部(如 GUI 库)获取的 void* 转换为恰当类型的指针。

R404：函数接口中不应使用内置数组类型，即如果一个函数参数是指针，那么它必须指向单个元素。如果希望传递数组，应使用 Array_ref。

原因：数组只能以指针方式传递，而元素数目无法附着其上，只能分开传递。而且，隐式的数组到指针的转换和派生类到基类的转换会引起内存错误。

类原则

R500：对于没有共有数据成员类，用 class 声明。对没有私有数据成员类，用 struct 声明。不要定义既有共有数据成员，又有私有数据成员类。

原因：清晰性。

r501：如果类包含析构函数或者指针/引用类型的成员，必须为其定义或禁止(即不能使用默认的)拷贝构造函数和拷贝赋值运算符。

原因：析构函数通常会释放资源。对于具有析构函数或指针和引用类型的类，默认拷贝语义几乎不可能“做正确的事”。

R502：如果类包含虚函数，那么它必须具有虚析构函数。

原因：虚函数可以通过基类接口来使用，通过基类接口访问对象的函数可能会删除对象，派生类必须有某种机制(析构函数)来进行清理工作。

r503：接受单一参数的构造函数必须显式声明。

原因：避免奇怪的隐式类型转换。

硬实时原则

R800：不应使用异常。

原因：异常不可预测。

R801：new 只能在初始化时使用。

原因：不可预测。

例外：可以用定址的 new 从栈中分配内存。

R802：不应使用 delete。

原因：不可预测，可能会引起碎片问题。

R803：不应使用 dynamic_cast。

原因：不可预测（假定是用普通方法实现的）。

R804：不应使用标准库容器，std::array 除外。

原因：不可预测（假定是用普通方法实现的）。

关键系统原则

R900：递增和递减运算不能作为子表达式。

例子：

```
int x = v[++i];    // violation
```

例子：

```
++i;
int x = v[i];      // OK
```

原因：可能会被漏掉。

R901：代码不应依赖于算术表达式优先级之下的优先级规则。

例子：

```
x = a*b+c; // OK
```

例子：

```
if (a<b || c<=d)    // violation: parenthesize (a<b) and (c<=d)
```

原因：C/C++ 基础较差的程序员写出的代码中常常会有优先级混乱的情况。

上面列出的编码原则并未按顺序编号，这样可以随时加入新的原则，而不必更改已有的编号，也不会破坏分类。编码原则常常以编号被人熟知，改变这些编号会受到用户的反对。

25.6.3 实际编码规范

已经有很多 C++ 编码规范，大多数是公司所有，并未广泛使用。在很多情况下，这对程序员来说可能是好事，也许只有这些公司的程序员除外。下面列出了一些对程序设计有帮助的编码规范，当然前提是将它们应用在恰当的领域：

Henricson, Mats, and Erik Nyquist. *Industrial Strength C++ : Rules and Recommendations*. Prentice Hall, 1996. ISBN 0131209655. 这是一个电信公司编制的规范。不幸的是，其中的编码原则有些过时了，因为这本书的出版日期早于 ISO C++ 标准，特别是它没有将模板纳入讨论范围。以现在的眼光来看，这本书中的编码原则都应该重写了。

Lockheed Martin Corporation. “Joint Strike Fighter Air Vehicle Coding Standards for the System Development and Demonstration Program.” 文档编号 2RDU00001 Rev C. 2005 年 12 月。俗称“JSF++”，这是洛克希德-马丁航空公司为飞机软件所编制的规范。编制和使用这个规范的人，是那些正在编写人类生活必不可少的软件的程序员。请参考 www.research.att.com/~

bs/JSF-AV-rules.pdf。

Programming Research. High-integrity C++ Coding Standard Manual 2.4 版。请参考 www.programmingresearch.com。

Sutter, Herb, and Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guide-lines, and Best Practices*. Addison-Wesley, 2004. ISBN 0321113586。本书很大程度上可以看做“元编码规范”，即它并不是定义特定的编码原则，而是为设计原则指出方向：什么是好的原则及为什么。

注意，并不是说阅读了以上书籍，你就不必再去了解实际的应用领域、程序设计语言以及相关的程序设计技术了。对于大多数应用领域，当然也包括嵌入式程序设计，你还是需要了解操作系统和硬件体系结构。如果你需要使用 C++ 进行低层程序设计，请查阅 ISO C++ 委员会关于性能的报告 (ISO/IEC TR 18015, www.research.att.com/~bs/performanceTR.pdf)。提及“性能”，他们/我们主要是指“嵌入式程序设计”。

语言方言和专有语言在嵌入式领域是很常见的，但只要条件允许，请使用标准语言（如 ISO C++）、标准工具和标准库。这会使你的学习曲线更短，而且可能延长你的工作成果的寿命。

简单练习

1. 编译、运行下面的程序：

```
int v = 1; for (int i = 0; i < sizeof(v)*8; ++i) { cout << v << ' '; v <<= 1; }
```

2. 将 v 改为 unsigned int 类型，重新编译、运行程序

3. 使用十六进制常量定义 short unsigned int 变量，使其值：

- 所有位都置位（置为 1）。
- 最低有效位置位。
- 最高有效位置位。
- 最低字节所有位都置位。
- 最高字节所有位都置位。
- 每隔一位置位（最低有效位置为 1）。
- 每隔一位置位（最低有效位置为 0）。

4. 将上题中每个数以十进制形式和十六进制形式输出。

5. 用位运算（|、&、<<），只用常量 1 和 0，重做第 3、4 题。

思考题

- 什么是嵌入式系统？给出十个例子，至少有三个是本章未提及的。
- 嵌入式系统的特殊之处在哪里？给出常见的五点。
- 给出嵌入式系统中可预测性的定义？
- 为什么嵌入式系统的维修很困难？
- 为什么出于性能的考虑进行系统优化是个糟糕的主意？
- 为什么我们更倾向于使用高层抽象而不是低层代码？
- 什么是瞬时错误？为什么我们特别害怕这种错误？
- 如何设计具有故障恢复能力的系统？
- 为什么我们无法防止所有故障？
- 什么是领域知识？给出一些应用领域的例子。
- 为什么对于嵌入式系统程序设计来说领域知识是必要的？
- 什么是子系统？给出一些例子。
- 从 C++ 语言的角度，存储可以分为哪三类？

14. 什么情况下你会使用动态内存分配?
15. 为什么在嵌入式系统中使用动态内存分配通常是不可行的?
16. 什么情况下在嵌入式系统中使用 `new` 是安全的?
17. 在嵌入式系统中使用 `std::vector` 的潜在问题是什么?
18. 在嵌入式系统中使用异常的潜在问题是什么?
19. 什么是递归函数调用? 为什么一些嵌入式系统程序员要避开它? 替代方法是什么?
20. 什么是内存碎片?
21. 什么是垃圾收集器(在程序设计中)?
22. 什么是内存泄漏? 它为什么会导致错误?
23. 什么是资源? 请举例。
24. 什么是资源泄漏? 如何系统地预防?
25. 为什么不能将对象从一个内存位置简单地移动到另一个位置?
26. 什么是栈?
27. 什么是存储池?
28. 为什么栈和存储池不会导致内存碎片?
29. 为什么 `reinterpret_cast` 是必要的? 它又会导致什么问题?
30. 指针作为函数参数有什么危险? 请举例。
31. 指针和数组可能引起什么问题? 请举例。
32. 在函数接口中, 可以用什么机制替代(指向数组的)指针参数?
33. “计算机科学第一定律”是什么?
34. 位是什么?
35. 字节是什么?
36. 通常一个字节有多少位?
37. 位运算有哪些?
38. 什么是“异或”运算? 它有什么用处?
39. 如何描述位序列?
40. 字中位的习惯编号次序是怎样的?
41. 字中字节的习惯编号次序是怎样的?
42. 什么是字?
43. 通常一个字中有多少位?
44. `0xf7` 的十进制值是多少?
45. `0xab` 的位序列是什么?
46. `bitset` 是什么? 你什么情况下需要使用它?
47. `unsigned int` 和 `signed int` 的区别是什么?
48. 什么时候使用 `unsigned int` 比 `signed int` 更好?
49. 如果需要处理的元素数目非常巨大, 如何设计一个循环?
50. 如果将 `-3` 赋予一个 `unsigned int` 变量, 它的值会是什么?
51. 我们为什么要直接处理位和字节(而不是处理上层数据类型)?
52. 什么是位域?
53. 位域的用途是什么?
54. 加密是什么? 为什么要加密?
55. 能对照片进行加密吗?
56. `TEA` 表示什么?
57. 如何以十六进制输出一个数?
58. 编码规范的目的是什么? 列出几条需要编码规范的原因。

59. 为什么没有一个普适的编码规范?
60. 列出一些好的编码规范应该具备的特点。
61. 编码规范如何会起到不好的效果?
62. 列出至少十条你认可(发现有用)的编码规范。解释它们为什么有用。
63. 我们为什么要避免使用字母全部大写的标识符?

术语

地址	加密	存储池	位	异或
可预测性	位域	小设备	实时	bitset
垃圾收集器	资源	编码规范	硬实时	软实时
嵌入式系统	泄漏	unsigned		

习题

1. 如果你还没有做本章中的“试一试”练习,现在做一下。
2. 为 0~9 的数字创建一个对应单词表,使得所有十六进制数字都能用英文字母(串)表示。如用 *o* 表示 0,用 *l* 表示 1,用 *to* 表示 2,等等。这样,十六进制数能够拼成像单词的形式,如 0xF001 拼写为 Fool, 0xBEEF 拼写为 Beef。在提交之前,小心地去除单词表中的粗话。
3. 用以下位模式初始化一个 32 位有符号整数,并打印出结果:全 0、全 1、1 和 0 交替(最高有效位为 1)、0 和 1 交替(最高有效位为 0)、两个 1 和两个 0 交替(110011001100...)、两个 0 和两个 1 交替(001100110011...)、全 1 字节和全 0 字节交替(最高字节为全 1)、全 0 字节和全 1 字节交替(最高字节为全 0)。对 32 位无符号数重做本题。
4. 为第 7 章的计算器程序添加位运算 &、|、^ 和 ~。
5. 编写一个无限循环,观察执行效果。
6. 编写一个不易察觉的无限循环。如果设计出的循环未能无限执行下去只是因为耗尽了某种系统资源,也可认为达到了本题的要求。
7. 输出 0~400 这些数值的十六进制形式,输出 -200 到 200 这些数值的十六进制形式。
8. 输出你的键盘上的每个字符的数值。
9. 不使用任何标准头文件(如 <limits>),也不借助任何文档,计算在你的系统中,一个 int 包含多少位,并确定 char 是有符号的还是无符号的。
10. 仔细分析 25.5.5 节中关于位域的例子程序。编写程序,初始化一个 PPN,然后读取并输出每个位域的值,接着改变每个位域的值(可以向每个位域赋值)并输出结果。改用一个 32 位无符号数存储 PPN,并使用位运算(参见 25.5.4 节)访问每个域,重做此题。
11. 重做上题,将二进制位保存在 `bitset<32>` 中。
12. 对 25.5.6 节中的例子,解密密文,输出明文。
13. 利用 TEA(参见 25.5.6 节)实现两台计算机之间的“保密”通信。最低限度要实现安全的 E-mail。
14. 实现一个简单 vector,能保存至多 *N* 个元素,存储空间从一个存储池中分配。测试 $N=1000$,元素类型为整数的情况。
15. 测试用 new 分配 10 000 个对象所花费的时间(参见 26.6.1 节),对象大小为 1 字节到 1000 字节之间的随机值,然后测试用 delete 释放这些对象所花费的时间。测试两次,第一次按分配的逆序进行释放,第二次按随机顺序进行释放。然后,测试从存储池中分配 10 000 个大小固定为 500 字节的对象的时间和释放它们的时间。接着测试从栈中分配 10 000 个大小为 1 字节到 1000 字节之间随机数的对象的时间和逆序释放它们的时间。比较测试结果。每个测试至少重复 3 次以确保结果是一致的。
16. 给出 20 条编码风格方面的原则(不要简单地复制 25.6 节中的内容)。将这些原则应用到你最近编写的一个 300 行以上的程序中。每应用一条原则,就为其编写一个简短(一或两行)的注释。在这个过程中,你是否发现代码中有错误?代码是否变得更清晰了?还是有些部分更不清晰了?根据这些结果修改编码原则。

17. 在 25.4.3 节和 25.4.4 节中我们提供了一个 `Array_ref` 类，宣称能以简单、安全的方式访问数组元素。特别是我们宣称它能正确处理继承。尝试用 `Array_ref < Shape* >` 以不同方式将一个 `Rectangle*` 放入 `vector < Circle* >` 中，不引起任何类型转换或其他行为不确定的操作。这应该是不可能办到的。



附言

那么，嵌入式程序设计基本上就是“摆弄位”吗？完全不是这样，特别是当你有意减少位运算，以免它成为影响正确性的潜在问题时。但是，在系统某些地方，我们不得不处理位和字节，问题只是在哪里，如何使用而已。在大多数系统中，低层代码可以也应该局部化，不应使位运算遍布整个程序。我们接触过的最有意思的系统中有许多都是嵌入式系统，而一些最有意思、最有挑战性的程序设计工作也是属于这个领域。

第26章 测试

“我只证明代码的正确性，不做测试。”

——Donald Knuth

本章介绍正确性相关的测试及设计技术。这是一个非常大的题目，我们在这一章中只能对它浅尝辄止。本章重点介绍一些单元测试的思想和技术，所谓单元测试，就是针对函数和类等程序单元进行测试。我们会讨论如何使用接口，以及如何选择测试。我们将着重介绍通过系统设计来简化测试工作，及在软件开发的早期就开展测试工作的重要性。我们还会简单介绍程序的正确性和处理性能问题方面的内容。

26.1 我们想要什么

让我们做一个简单的实验：写一个二分搜索程序。现在就写，不要等到本章的结束，不要等到下一节。亲自动手是非常重要的，就是现在！二分搜索是一种用于有序序列搜索的方法，开始时先访问序列中间元素：

- 如果中间元素等于我们要搜索的元素，结束搜索。
- 如果中间元素小于我们要搜索的元素，我们继续使用二分方法搜索右半部分。
- 如果中间元素大于我们要搜索的元素，我们继续使用二分方法搜索左半部分。
- 结果显示了搜索是否成功，这里可以使用一些允许我们修改元素的工具，例如，下标、指针或迭代因子。

可以使用小于运算(<)进行比较(排序)操作。按照你的喜好，可以使用任何一种数据结构，任何一种函数调用规范，以及任何一种返回结果的方法，但是一定要亲手编写这个程序。在本例中，使用他人的代码是会起反作用的，即便你列明了出处。特别要注意的是，不要使用标准库算法(binary_search 或 equal_range)，虽然在大多数情况下，它是你的首选。多花点时间在这上面吧。

现在你已经完成了你自己的二分搜索函数了。如果没有，请返回上一段。如何确定你的搜索函数是正确的呢？如果还无法确定的话，你可以先写下你认为代码正确的理由。如何相信你的理由呢？是否有部分参数可能会有问题？

这是一段简单、平凡的代码。它使用了最经典的算法来实现。你的编译器用了20万行代码，你的操作系统包含了1000万到5000万行代码，你下一次度假或开会时搭乘的飞机上的安全系统包含了50万到200万行代码。这会让你感到舒服一点吗？你在二分搜索函数中使用的技术是如何应用到这类大型实际软件中去的呢？

令人好奇的是，虽然包含如此复杂的代码，大多数大型软件在绝大部分时间都能正常工作。当然，我们也不会把以游戏为主的消费型PC上运行的软件当做“关键系统”。对我们来说更为重要的是，对安全性要求严格的软件几乎需要在任何时刻都能正常运行。我们想不回去十年中发生过因软件问题造成飞机或汽车事故的案例。至于金额为0.00美元的支票导致银行软件严重混乱的故事，也已经非常久远了，这些问题基本上不会再发生了。但是，软件毕竟是像你我这样的人编写的。你很清楚自己是会犯错的，事实上我们都会犯错误。那么如何让软件不会出错呢？

最基本的答案是：“我们”已经知道如何用不可靠的组件构建可靠的系统。我们尽力让每一个程序、每一个类、每一个函数都正确，但在最初总是会出错的。于是我们调试、测试、重新设计程序，找出并排除尽可能多的错误。然而，在任何复杂系统中，还是会有错误隐藏其中。我们知道错误存在，但我们无法找到它们。或者说，在有限的时间内，在可以投入的人力、物力条件下，我们无法找到这些错误。于是，我们继续重新设计系统，来解决这些不可预测或“不可能”的错误。最终得到的可能是一个非常可靠的系统。但请注意，即使是这些非常可靠的系统，其中也隐藏着错误，只不过大多数情况下系统都能工作正常，只是偶尔运行状况不如我们预期。但是，这类系统不会崩溃，并且提供的服务总能满足最低需求。例如，在通话请求异常高的时候，电话系统可能不能满足所有通话请求，但它仍然可以提供许多连接服务。

现在，我们可以上升到哲学层面，讨论一下我们所猜测和顾忌的不可预见的错误是否是真正的错误，是否真的会发生。不过我们先不考虑这个问题，我们要做的是更有价值的事情：弄明白如何才能使我们的系统更加稳定。

26.1.1 说明

测试是一个很大的主题。很多人都在研究测试应该怎样做，不同的工业和应用领域有着不同的习惯和标准。这很自然，对于视频游戏软件和航空系统软件，你所需要的可靠性标准显然是不一样的。但这种情况会导致术语和工具的混乱。本章介绍的内容可以作为你个人项目进行测试的思想源泉，如果你参与大型系统的测试，也可以从本章汲取思想。大型系统的测试包括了各种测试工具和组织结构的组合，这些内容不在本章的讨论范围之内。

26.2 程序正确性证明

等一下！为什么我们要为测试问题忙得团团转，为什么不能直接证明程序的正确性呢？正如Edsger Dijkstra一针见血地指出的那样，“测试只能揭示错误的存在，而不能证明不存在错误”。这引出了对程序正确性证明的一个明显的要求：“像数学家证明数学定理那样”。

不幸的是，证明一个普通程序的正确性是现有研究所不能解决的（一些非常特殊的应用领域除外），而且证明本身也会包含错误（就像数学家也会出错一样），整个程序证明领域也是一个非常高深的研究方向。因此，我们要尽可能使程序更加结构化，以便能对其进行推理，使我们能确信它是正确的。但是，我们还是要进行测试（参见26.3节），并更好地组织代码，使之具备错误恢复能力，能应对剩余未发现的错误（参见26.4节）。

26.3 测试

在5.11节中，我们说过“测试是一种系统地查找错误的方法”。下面，让我们来看一下相关的技术。

一般来说，人们把测试分为单元测试（unit testing）和系统测试（system testing）两种。“单元”是指函数和类等程序的组成部分。当我们对单元进行独立测试的时候，一旦错误发生，我们就能确定在哪里寻找错误——错误一定是在我们所测试的单元中（或者在用来引导测试的代码中）。与之相对，系统测试只能告诉我们系统中存在错误。一种典型情况是，当我们完成单元测试后，在系统测试中发现了错误，那么可能是单元之间的交互出现了问题。与独立单元内的错误相比，这类错误更难查找，也更难修复。

显然，一个单元（如一个类）可以包含多个小单元（如函数或其他类）。一个系统（如电子商务

系统)也可以包含多个子系统(如数据库、GUI、网络系统、订单确认系统)。因此,单元测试和系统测试的区别不像你想象得那么清晰,但有一个一般性的策略:做好自己编写的单元的测试,这样既节省了自己的时间,也减少了最终用户的烦恼。

关于测试的一种看法是:任何复杂系统都是由许多单元组成的,这些单元又是由更小的单元组成的。因此,我们从最小的单元开始测试,然后再依次测试更大的单元,直到整个系统测试完毕为止。也就是说,“系统”是最大的单元(除非我们用它来构建更大的系统)。

因此,我们首先考虑的是如何测试一个单元(如函数、类、嵌套类或模板)。测试有白盒测试(你可以看到被测单元的实现细节)和黑盒测试(你只能看到被测单元的接口)之分。我们不会花很多精力区分这两种方式,你应该尽一切办法了解被测单元的细节。但要注意的是,可能有人随后修改被测单元,因此,不要依赖任何在单元接口中无法确定的信息。实际上,测试的基本思想是向接口发送被测单元可以接受的任何输入,然后观察其反应是否正确。

需要注意的是,因为被测单元的代码可能会被修改(你或其他人),这就需要进行回归测试。基本上,只要你修改了代码,就必须重新进行测试,以保证你的修改没有造成破坏。因此,在代码升级后,必须重新进行单元测试,在整个系统提交前(或者你自己使用前),也要重新进行完整的系统测试。

这种对系统的完整测试通常称为回归测试(regression testing),因为这种测试通常包括对以前发现的错误的再次检查,以避免代码修改将错误重新引入。如果旧的错误仍然存在,系统就“回归”了,需要再次修正错误。

26.3.1 回归测试

收集在过去测试中有助于发现错误的测试用例,是为系统建立有效测试集的主要工作。如果有用户在使用系统的话,他们会将错误信息发送给你。千万不要将这些错误报告丢弃,要用错误跟踪系统来确保这一点。因为,一个错误报告表明:或者存在一个系统错误,或者存在一个用户使用问题,两者都是有用的。

通常,错误报告会包括许多无关的信息,我们的第一项任务就是将所报告的问题局限在程序的最小范围内。这经常要去除掉大部分代码,一般我们会试图去掉库的使用和不会导致错误的应用程序代码。找出最小被测对象有助于我们在系统代码中确定错误区域,这一被测对象将被提交做回归测试。找出最小测试对象的方法是不断去除无关代码直到错误出现为止,然后将最后一次去除的代码重新加入。不断持续这一过程直到不能再删除代码为止。

仅仅运行上百个(或上万个)来自错误报告的测试用例并不能表明测试的系统性,我们真正要做的是系统地利用用户和开发人员的经验。回归测试用例集就体现了开发者的群体记忆。对一个大系统来说,我们不能简单地依靠开发人员来了解设计和实现的细节。回归测试用例集就可以确保系统的变化不会脱开发者和用户所认可的范围。

26.3.2 单元测试

好吧,已经说得够多了!让我们来尝试一个实际的例子吧:测试一个二分搜索。下面的描述来自 ISO 标准(参见 25.3.3.4 节):

```
template<class ForwardIterator, class T>
bool binary_search(ForwardIterator first, ForwardIterator last,
                  const T& value);

template<class ForwardIterator, class T, class Compare>
bool binary_search(ForwardIterator first, ForwardIterator last,
                  const T& value, Compare comp);
```

要求: $[first, last)$ 之间所有元素 e 按表达式 $e < value$ 和 $!(value < e)$ 或 $comp(e, value)$ 和 $comp(value, e)$ 划分。并且, 对 $[first, last)$ 之间所有元素 e 来说, $e < value$ 意味着 $!(value < e)$, $comp(e, value)$ 意味着 $!comp(value, e)$ 。

返回: 如果位于区间 $[first, last)$ 内的一个迭代器 i 满足如下条件: $!(*i < value) \&\& !(value < *i)$ 或 $comp(*i, value) == false \&\& comp(value, *i) == false$, 函数返回真。

复杂度: 最多 $\log(last - first) + 2$ 次比较。

没有经验的人很难读懂上述形式化定义(好吧, 只是半形式化的)。但是如果你已经真正完成了本章开头我们所强烈建议的二分搜索编程练习的话, 你就会对如何实现二分搜索并测试它有一些很好的想法。这个(标准)版本接受三个参数: 两个前向迭代器(参见 20.10.1 节)和一个数值。如果数值出现在两个迭代器限定的范围内, 函数就返回真。两个迭代器必须对应一个有序序列。比较(排序)操作通过运算符“ $<$ ”完成。我们可以为 `binary_search` 增加一个参数——比较操作函数, 这样就可以用用户指定的任意比较操作进行二分搜索了, 我们将此作为练习。

在这里, 我们只处理编译器不能发现的错误。因此, 类似下面的问题不再考虑:

```
binary_search(1,4,5);           // error: an int is not a forward iterator
vector<int> v(10);
binary_search(v.begin(),v.end(),"7"); // error: can't search for a string
                                         // in a vector of ints
binary_search(v.begin(),v.end());   // error: forgot the value
```

我们应该如何系统地测试 `binary_search()` 呢? 显然, 我们不可能测试所有可能的参数。因为可能的参数值和类型的组合的数目会是一个非常巨大的数字! 因此, 我们必须精挑细选测试用例。我们需要一些选择标准:

- 很可能导致错误的测试(找出大多数错误)。
- 可能导致严重错误的测试(找出可能导致最坏结果的错误)。

这里的“严重错误”是指那些会导致最坏结果的错误。通常, 这是一个模糊的概念。但对特定程序来说, 就可能非常明确。例如, 孤立考虑二分搜索的话, 所有错误的严重程度都差不多。但是, 如果 `binary_search` 是用于一个大程序, 而该程序会将所有计算结果都仔细检查两遍的话, “返回一个错误结果”要比“陷入死循环什么也不返回”好得多。当查找这类错误时, 将 `binary_search` “骗”入一个死循环(或者是非常长的循环), 要比“骗”它返回一个错误结果花费多得多的力气。注意, 我们使用了“欺骗”一词。与其他工作相比, 测试就是一种发挥我们的创造性思维, 来达到“如何让这个程序运行不正常?”目的的活动。因此, 最好的测试人员不但要有系统性思维, 还要非常“狡猾”(当然, 要有正当的理由)。

26.3.2.1 测试策略

我们应该如何破坏 `binary_search` 的正常运行呢? 首先, 我们要检查 `binary_search` 的要求, 即它对于输入的假设是怎样的。不幸的是, 从测试者的观点来看, `binary_search` 明确声明 $[first, last)$ 必须是有序序列, 也就是说, 确保这一点是调用者的责任。因此, 我们不能通过输入无序序列, 或者 $last < first$ 来破坏 `binary_search`, 那是不公平的。注意, 在 `binary_search` 的要求中并没有说明, 如果我们给定的输入不满足条件的話, 它会如何做。在 ISO 标准的其他地方, `binary_search` 声明: 对于不满足要求的输入, 它可能会抛出一个异常, 但这不是强制的。我们要记住这些事实, 当测试其他程序对 `binary_search` 的使用方式时, 这些事实是很有用的: 调用者给出不满足函数要求的输入, 显然可能成为错误之源。

我们设想 `binary_search` 可能出现下列错误:

- 不返回(例如, 无限循环)

- 崩溃(例如,错误的引用,无限递归)
- 未找到值,即使该值确实在序列中
- 找到了值,但该值并不在序列中

此外,我们还要记住下列序列会给用户错误“可乘之机”:

- 序列未排序(例如, {2,1,5,-7,2,10})。
- 序列不合法(例如, binary_search(&a[100], &a[50], 77))

我们只不过是简单调用 binary_search(p1, p2, v) 而已,为什么函数还会出现错误呢(均为测试者发现的错误)?一般来说,通常是“特殊情况”导致错误发生。特别是,当测试对象是处理序列的程序时,可以从序列开始和末尾入手构造“特殊序列”。另外,我们总是应该对空序列进行测试。让我们先来看一些有序的整型数组:

```
{1,2,3,5,8,13,21}    // an "ordinary sequence"
{}                    // the empty sequence
{1}                  // just one element
{1,2,3,4}            // even number of elements
{1,2,3,4,5}          // odd number of elements
{1,1,1,1,1,1,1}      // all elements equal
{0,1,1,1,1,1,1,1,1,1} // different element at beginning
{0,0,0,0,0,0,0,0,0,0,1} // different element at end
```

也可以用程序来生成一些测试序列:

```
vector<int> v1;
for (int i=0; i<100000000; ++i) v.push_back(i); // a very large sequence
```

- 一些元素个数随机的序列
- 一些由随机数组成的序列(仍是有序的序列)

这不是我们预期的那种系统测试。毕竟,我们只是“挑拣”了一些序列。但是,这里用到了一些处理数据集时很有用的一般性原则,包括:

- 空集
- 小数据集
- 大数据集
- 极限分布的数据集
- 序列末尾处可能发生问题的数据集
- 包含重复元素的数据集
- 包含奇数和偶数个元素的数据集
- 由随机数组成的数据集

使用随机数序列的目的是看看是否能幸运地发现一些我们没有考虑到的错误。这是一种蛮力技术,但是能节省我们的时间。

为什么要使用“奇数/偶数”个元素的序列呢?这是因为很多算法都会用划分的方法把输入序列分为两部分,而程序员可能只考虑了奇数情况或偶数情况。更一般的问题是,当我们划分一个序列的时候,划分会成为一个子序列的末尾。正如我们所知,序列的末尾往往是错误容易发生的地方。

一般来说,在设计测试用例时我们应该重点考虑:

- 极限情况(大的、小的、奇异分布的输入等)
- 边界条件(边界附近的任何情况)

“极限情况”、“边界条件”具体是什么含义,取决于我们所测试的实际程序。

26.3.2.2 一个简单的测试

我们可以进行两类测试：应该搜索成功的测试（例如，搜索在序列中确实存在的值）；应该搜索失败的测试（例如，搜索一个空集）。对每一组输入序列，我们都会构造应该成功和应该失败的测试用例。我们将从最简单和最明显的用例开始，然后逐步改进，直至找到对 `binary_search` 来说足够好的测试用例：

```
int a[] = { 1,2,3,5,8,13,21 };
if (binary_search(a,a+sizeof(a)/sizeof(*a),1) == false) cout << "failed";
if (binary_search(a,a+sizeof(a)/sizeof(*a),5) == false) cout << "failed";
if (binary_search(a,a+sizeof(a)/sizeof(*a),8) == false) cout << "failed";
if (binary_search(a,a+sizeof(a)/sizeof(*a),21) == false) cout << "failed";
if (binary_search(a,a+sizeof(a)/sizeof(*a),-7) == true) cout << "failed";
if (binary_search(a,a+sizeof(a)/sizeof(*a),4) == true) cout << "failed";
if (binary_search(a,a+sizeof(a)/sizeof(*a),22) == true) cout << "failed";
```

这个测试虽然有些重复和繁琐，但却是一个不错的开端。实际上，许多简单的测试集与这个例子一样，就是一个长长的调用序列。这种方法最大的优点就是简单。即使是一个新手，也能够测试集中加入新的测试用例。但是，我们通常还可以做得更好。例如，当某个测试失败时，这个程序没有告诉我们哪个测试用例失败了。这是不可接受的，改进如下：

```
int a[] = { 1,2,3,5,8,13,21 };
if (binary_search(a,a+sizeof(a)/sizeof(*a),1) == false) cout << "1 failed";
if (binary_search(a,a+sizeof(a)/sizeof(*a),5) == false) cout << "2 failed";
if (binary_search(a,a+sizeof(a)/sizeof(*a),8) == false) cout << "3 failed";
if (binary_search(a,a+sizeof(a)/sizeof(*a),21) == false) cout << "4 failed";
if (binary_search(a,a+sizeof(a)/sizeof(*a),-7) == true) cout << "5 failed";
if (binary_search(a,a+sizeof(a)/sizeof(*a),4) == true) cout << "6 failed";
if (binary_search(a,a+sizeof(a)/sizeof(*a),22) == true) cout << "7 failed";
```

假定最终我们会进行数十个测试，有没有这种改进就会大不一样了。对于真实系统，我们经常要进行几千个测试。因此，准确定位哪个测试出错是非常必要的。

在继续讨论之前，请注意上面例子中所体现出的（半系统化）测试技术：我们从序列的末尾和“中部”取一些值作为要搜索的值，这些测试用例应导致搜索成功。我们当然可以将该序列的所有值逐一作为输入，但这显然是不现实的。对于导致搜索失败的测试用例，我们从序列两端和中部各选择一个值（不在序列中的值）。当然，这也不是一种系统化的测试方法，但这种测试用例构造模式是一种十分常用的技术，它在测试数值序列或数值范围类程序时非常有用。

这些初步的测试有什么问题么？

- 重复编写相同的代码。
- 手工设定测试编号。
- 输出信息很少（用处也不大）。

经过仔细考虑后，我们决定把测试用例保存在文件中。每个测试都包含一个唯一的标签、一个要搜索的值、数值序列以及期望的计算结果。例如，

```
{ 277 { 1 2 3 5 8 13 21 } 0 }
```

这个测试的编号是27。它在序列{1,2,3,5,8,13,21}中查找7，期望运行结果是0（即失败）。我们为什么要把测试用例保存到文件中，而不是把它们硬编码到程序中呢？是的，对于这个例子来说，我们可以直接输入测试用例，放在程序中。但是，在程序中放入大量数据，无疑会使程序变得杂乱无章。而且，我们经常会用程序生成测试用例，而不是手工编写。计算机生成的测试用例一般都保存在数据文件中，无法直接放在程序之中。现在，我们可以编写一个使用各种不同测试用例文件的测试程序了：

```

struct Test {
    string label;
    int val;
    vector<int> seq;
    bool res;
};

istream& operator>>(istream& is, Test& t); // use the described format

int test_all()
{
    int error_count = 0;
    Test t;
    while (cin>>t) {
        bool r = binary_search( t.seq.begin(), t.seq.end(), t.val);
        if (r !=t.res) {
            cout << "failure: test " << t.label
                << " binary_search: "
                << t.seq.size() << " elements, val==" << t.val
                << " -> " << t.res << "\n";
            ++error_count;
        }
    }
    return error_count;
}

int main()
{
    int errors = test_all();
    cout << "number of errors: " << errors << "\n";
}

```

下面是我们所使用的部分输入序列：

```

{1.11{1 2 3 5 8 13 21}1}
{1.25{1 2 3 5 8 13 21}1}
{1.38{1 2 3 5 8 13 21}1}
{1.421{1 2 3 5 8 13 21}1}
{1.5-7{1 2 3 5 8 13 21}0}
{1.64{1 2 3 5 8 13 21}0}
{1.722{1 2 3 5 8 13 21}0}

{21{}0}

{3.11{1}1}
{3.20{1}0}
{3.32{1}0}

```

在这里可以看出，为什么我们将标签定义为字符串类型，而不是数值类型：可以更灵活地为测试“编号”——本例中使用了带小数的标签，来区分同一个序列之上的不同测试。我们还可以使用更复杂的格式，来避免在测试数据文件中反复给出同一个序列。

26.3.2.3 随机序列

在选择测试数据的时候，我们会尽力击败程序编写者（常常就是我们自己），会重点在那些可能隐藏有错误的区域选择数据（例如，复杂的条件序列、序列末尾处、循环序列等）。但是，由于通常我们就是程序的编写者，我们在编写和调试代码时已经考虑过这些因素了。因此，我们在设计测试方案时很可能重复编写程序时所犯的逻辑错误，这导致一些重要问题被忽略。这也是为什么要让与开发人员无关的另一些人参与到测试方案设计中的原因之一。有一种技术有时会对

解决这一问题有所帮助：简单地生成（许多）随机值。下面这个函数使用 `rand_int()`（参见 24.7 节）生成一个二分搜索测试用例，并输出到 `cout`：

```
void make_test(const string& lab, int n, int base, int spread)
    // write a test description with the label lab to cout
    // generate a sequence of n elements starting at base
    // the average distance between elements is spread
{
    cout << "{" << lab << " " << n << " {" ;
    vector<int> v;
    int elem = base;
    for (int i = 0; i < n; ++i) {    // make elements
        elem += rand_int(spread);
        v.push_back(elem);
    }

    int val = base + rand_int(elem - base);    // make search value
    bool found = false;
    for (int i = 0; i < n; ++i) {    // print elements and see if val is found
        if (v[i] == val) found = true;
        cout << v[i] << " ";
    }
    cout << "}" << found << " \n";
}
```

请注意我们没有用 `binary_search` 来检验随机数 `val` 是否在随机序列中。我们不能用待测程序来检查测试的正确性。

实际上，`binary_search` 并不特别适合用随机数序列进行蛮力测试。虽然我们对这些测试用例能否发现我们“手工构造”的测试用例未发现的错误持怀疑态度，但这些技术通常还是很管用的。不管怎么样，让我们先动手构造一些基于随机数的测试：

```
int no_of_tests = rand_int(100);    // make about 50 tests
for (int i = 0; i < no_of_tests; ++i) {
    string lab = "rand_test_";
    make_test(lab + to_string(i),    // to_string from §23.2

              rand_int(500),        // number of elements
              0,                    // base
              rand_int(50));        // spread
}
```

如果我们需要测试很多操作的累积效应，而一个操作的结果取决于之前的操作是如何处理的话。即系统是有状态的（参见 5.2 节），系统运行就是状态间的迁移。对于这种情况，基于随机数的测试用例特别有用。

基于随机数的测试用例对 `binary_search` 效果不是很明显，原因在于，对于同一个序列，不同搜索之间都是独立的。当然，这个结论的前提是假定 `binary_search` 的实现中没有犯致命的愚蠢错误，例如，对序列进行了修改。对此，我们可以设计一个更好的测试用例（参见习题 5）。

26.3.3 算法和非算法

上文以 `binary_search()` 为例介绍了一些简单的测试技术，`binary_search()` 是一个完全合乎格式要求的算法，它具有下列特点：

- 对输入数据有明确的要求。
- 明确描述了算法运行后对输入数据有什么影响（在本例中，没有影响）。
- 算法不依赖于显式输入之外的数据。
- 对于外部环境没有严格限制（例如，没有特殊的时间、空间和资源共享要求）。

它还包含了明显的前置和后置条件(参见 5.10 节)。换句话说,它是测试人员梦寐以求的。但是,我们不可能总是这么幸运:很多时候我们不得不测试用糟糕的英文和很多图表表示的混乱代码(这还是乐观估计)。

等一下,我们是否对混乱的程序逻辑有些放任了呢?在不能准确描述代码要做什么的情况下,我们如何能够讨论正确性和测试呢?但问题是,在软件开发、测试中,很多内容都很难用非常清晰的数学形式来描述。而且,很多时候虽然在理论上能够给出严谨的数学描述,但所需数学知识超出了编写和测试代码的程序员的能力。因此,在现实世界的实际条件以及时间的双重压力下,我们只好放弃准确描述被测程序的理想目标,而要面对现实:只要被测程序在(测试人员,很多时候就是我们自己)掌握之中就可以了。

假设你要测试一个混乱的函数代码,这里“混乱”是指:

- 输入:它对输入(隐式或显式的)的要求没有像我们希望的那样进行准确定义。
- 输出:它对输出(隐式或显式的)的要求没有像我们希望的那样进行准确定义。
- 资源:它所使用的资源(时间、内存、文件等)没有像我们希望的那样准确定义。

这里的“隐式或显式”表示我们不但要检查正式的参数和返回值,还要检查函数对全局变量、`iostream`、文件、空闲内存空间的分配等的影响。那么,我们要怎么做呢?首先,这类函数一般都很长,或者我们不能把它的需求和影响描述清楚。也许我们讨论的是一个有 5 页长的函数,或者它使用复杂且不明朗的“帮助函数”的方式。你可能会认为 5 页很长了,是的,这确实很长,但我们还会遇到更长的函数。而且不幸的是,这种事情经常会发生。

如果这是我们写的代码并且有足够的时间的话,首先要做的是把这些“混乱的函数”分割成为许多小函数。每个小函数都符合我们理想中的严格定义函数,然后首先测试这些小函数。但是,我们的目标是测试软件——即系统地找出尽可能多的错误——而不是修正找到的错误。

那么,我们要找什么呢?作为测试人员,我们的任务是找出错误。错误可能隐藏在哪呢?包含错误的代码有什么特点呢?

- 与“其他代码”微妙的相关性:因此我们可以检查全局变量、非常量引用参数、指针等的使用。
- 资源管理:查找内存管理(`new` 和 `delete` 操作)、文件使用、锁等。
- 查找循环:检查终止条件(例如 `binary_search()`)。
- `if` 和 `switch` 语句(也称为“分支语句”):查找这些程序中的逻辑错误。

让我们逐个举例说明。

26.3.3.1 相关性

考虑下面这个无意义的函数:

```
int do_dependent(int a, int& b)    // messy function
    // undisciplined dependencies
{
    int val;
    cin >> val;
    vec[val] += 10;
    cout << a;
    b++;
    return b;
}
```

在测试 `do_dependent()` 的时候,我们不能仅仅检查参数合法性,以及函数对参数做了什么运算。我们还要考虑函数所使用的全局变量 `cin`、`cout` 和 `vec`。在这个小函数中,对这些全局变量的使用方式很容易看清,但在实际程序中,这些细节往往会隐藏在大量代码中间。幸运的是,一些软件

可以帮助我们找出这种相关性。不幸的是,这一办法并不总是可行,也很难推广。假定没有分析软件能帮助我们,我们只能逐行检查代码,找出其中所有的相关性。

在测试 `do_dependent()` 的时候,我们需要考虑

- 它的输入:
 - `a` 的值
 - `b` 的值以及 `b` 所引用的整型值
 - `cin` 的输入值(存入 `val`)和 `cin` 的状态
 - `cout` 的状态
 - `vec` 的值,以及 `vec[val]` 的值
- 它的输出:
 - 返回值
 - `b` 所引用的整型值(我们对它做了增量操作)
 - `cin` 的状态(包括流状态和格式状态)
 - `cout` 的状态(包括流状态和格式状态)
 - `vec` 的状态(我们对 `vec[val]` 做了赋值操作)
 - 任何 `vec` 可能抛出的异常(`vec[val]` 可能越界)

这是一个很长的列表,实际上,它比函数本身都要长。这也可以解释为什么我们不喜欢全局变量,并且非常关注非常量引用(以及指针)。最理想的情况莫过于一个函数仅读入它的参数,计算结果只以返回值的形式给出:这样我们就能很容易地理解并测试这样的函数。

一旦确定了输入和输出,我们就可以回过头来看看 `binary_search()` 的例子。我们测试的方式是给出输入值(隐式或显式的输入),检查函数是否输出期望的结果(隐式或显式的)。对于 `do_dependent()`,我们需要从一个非常大的 `val` 值和负的 `val` 值开始,来看看它会输出什么。而且,看上去 `vec` 最好具备边界检查机制(否则我们可以很容易地构造出非常严重的错误)。当然,我们还要按照文档的说明检查所有的输入和输出。但对于这种混乱的函数来说,我们不要期望它会有清晰、完整和准确的说明。因此,我们只需考虑如何击破这个函数(即找到错误),然后开始询问什么是正确的。通常情况下,这样的测试和询问会导致代码的重新设计。

26.3.3.2 资源管理

考虑下面这个无意义的函数:

```
void do_resources1(int a, int b, const char* s) // messy function
// undisciplined resource use
{
    FILE* f = fopen(s, "r"); // open file (C style)
    int* p = new int[a];      // allocate some memory
    if (b <= 0) throw Bad_arg(); // maybe throw an exception
    int* q = new int[b];      // allocate some more memory
    delete[] p;              // deallocate the memory pointed to by p
}
```

在测试 `do_resources1()` 的时候,我们必须考虑每个申请到的资源是否都被妥善处理了,即是否每一个资源都被释放或者转交给了其他函数了。

在本例中,显然存在这些问题

- 名为 `s` 的文件没有关闭。
- 如果 `b <= 0` 或者第二个 `new` 发生异常的话,指针 `p` 指向的内存会发生泄漏。
- 如果 `0 < b` 的话,指针 `q` 指向的内存会发生泄漏。

此外,我们还要考虑打开文件操作可能会失败。为了显示这种糟糕的结果,我们故意使用了一种非常古老的编程风格(`fopen()`是 C 语言中的打开文件的标准方法)。为了使测试人员的工作更为简单,我们可以将代码改写如下:

```
void do_resources2(int a, int b, const char* s) // less messy function
{
    ifstream is(s);           // open file
    vector<int> v1(a);         // create vector (owning memory)
    if (b<=0) throw Bad_arg(); // maybe throw an exception
    vector<int> v2(b);         // create another vector (owning memory)
}
```

现在每一块内存空间都被一个能够自己释放内存的对象所拥有。考虑如何才能写出更简洁(更清晰)的函数,有时是思考测试方法的很好途径。19.5.2 节中的“分配到的资源都要进行初始化”(RAII)技术提供了一种解决资源管理问题的一般策略。

需要注意的是,资源管理不仅仅是检查每一块内存是否被释放。有时,我们会从其他地方接收到资源(例如作为参数),有时我们也会将资源传出函数(例如作为返回值)。在这种情况下,很难确定什么是正确的。考虑下面的函数:

```
FILE* do_resources3(int a, int* p, const char* s) // messy function
// undisciplined resource passing
{
    FILE* f = fopen(s,"r");
    delete p;
    delete var;
    var = new int[27];
    return f;
}
```

`do_resources3()`将打开的文件作为返回值是否正确呢?通过参数 `p` 传递给它的内存释放是否正确呢?此外,我们还偷偷改变了全局变量 `var`(显然它是一个指针)。基本上,将资源传进/传出是很常见也很有用的,但要知道资源传递是否正确,还要掌握一些资源管理策略方面的知识。谁拥有这些资源?谁将删除/释放这些资源?程序文档应该清楚、简洁地回答这些问题(通常只是我们的美好愿望)。不管怎样,资源的传递都是孕育错误的温床,当然这也是测试的重点之一。

需要注意的是,在上面的例子中,我们是如何通过使用全局变量(故意地)使资源管理复杂化的。当我们把各种可能的错误来源混杂在一起的时候,一切都会变得糟糕。作为程序员,我们要避免这一点。作为测试人员,我们要重点检查这种情况。

26.3.3.3 循环

当我们讨论 `binary_search()` 的时候,对循环结构进行了检查。基本上,大部分错误都是在循环结构中发生的:

- 在开始循环的时候,是否所有数据都正确地初始化了?
- 循环是否正确地终止了呢(经常是在最后一个元素出问题)?

下面是一个循环结构出错的例子:

```
int do_loop(vector<int>& v) // messy function
// undisciplined loop
{
    int i;
    int sum;
    while(i<=vec.size()) sum+=v[i];
    return sum;
}
```

这个例子有三处明显的错误。(哪三处?)此外,好的测试人员应该马上意识到对 `sum` 的加法运算

可能会导致溢出问题。

- 很多循环都包含数据处理, 因此当有大量输入数据时, 可能会产生某种溢出错误。

一个臭名昭著的循环错误是缓冲区溢出, 我们可以通过系统地询问两个关于循环的关键问题, 来捕获这类错误:

```
char buf[MAX];    // fixed-size buffer

char* read_line() // dangerously sloppy
{
    int i = 0;
    char ch;
    while(cin.get(ch) && ch!='\n') buf[i++] = ch;
    buf[i+1] = 0;
    return buf;
}
```

当然, 你不会如此编写代码! (为什么不? read_line() 有什么问题吗?) 但糟糕的是, 这种代码编写方法很常见, 而且还有许多变化形式, 例如:

```
// dangerously sloppy:
gets(buf);           // read a line into buf
scanf("%s", buf);    // read a line into buf
```

在文档中查阅 gets() 和 scanf() 的相关内容, 要像躲避瘟疫一样躲开这两个函数。这里“危险”的含义是: 这种缓冲区溢出是“黑客攻击”(即非法闯入计算机系统)的主要手段。现在很多编译器都会对 gets() 及其近亲给出警告信息, 原因就在于此。

26.3.3.4 分支

显然, 当必须做出选择的时候, 我们可能会做出错误的抉择。这就使得 if 和 switch 语句成为测试人员的好目标。有两个主要问题需要检查:

- 所有的可能性都被覆盖了吗?
- 操作是否与分支正确联系起来了?

考虑下面这个函数:

```
void do_branch1(int x, int y) // messy function
// undisciplined use of if
{
    if (x<0) {
        if (y<0)
            cout << "very negative\n";
        else
            cout << "somewhat negative\n";
    }
    else if (x>0) {
        if (y<0)
            cout << "very positive\n";
        else
            cout << "somewhat positive\n";
    }
}
```

其中最明显的错误是我们“忘记”了 x 是 0 的情况。当测试非 0 值的时候(或是测试正值和负值的时候), 0 经常被忘记或者错误地与其他情况混在一起(例如考虑负数的情况)。此外, 这个程序中还隐藏着一个很微妙(但不常见)的错误: $(x > 0 \ \&\& \ y < 0)$ 和 $(x > 0 \ \&\& \ y \geq 0)$, 从某种角度看, 它们是被颠倒了。这通常是在编辑程序时使用剪切-粘贴操作所导致的。

if 语句的使用方式越复杂, 就越可能出现这种错误。从测试人员的角度出发, 我们应该检查

代码并确保所有分支都被测试。对于 `do_branch1()` 来说, 一个明显的测试集是

```
do_branch1(-1,-1);
do_branch1(-1, 1);
do_branch1(1,-1);
do_branch1(1,1);
do_branch1(-1,0);
do_branch1(0,-1);
do_branch1(1,0);
do_branch1(0,1);
do_branch1(0,0);
```

基本上, 这是一种蛮力测试方法, 通过“遍历所有可能”来查找错误。由于我们已经注意到 `do_branch1()` 使用“`<`”和“`>`”检测非 0 值, 因此采用了这一方法。此外, 为了检查 `x` 为正值时的可能错误, 我们还需要把每个调用与期望的正确结果结合起来。

处理 `switch` 语句的方法与 `if` 语句基本上是一样的。

```
void do_branch1(int x, int y) // messy function
    // undisciplined use of switch
{
    if (y<0 && y<=3)
        switch (x) {
            case 1:
                cout << "one\n";
                break;
            case 2:
                cout << "two\n";
            case 3:
                cout << "three\n";
        }
}
```

这里, 我们犯了 4 个错误:

- 我们对错误的变量进行了范围检查(应该是 `y` 而不是 `x`)。
- 对 `x == 2`, 我们忘记了 `break` 语句, 这会导致错误的操作。
- 我们忘记了 `default` 情况(认为在 `if` 语句中已经考虑了这种情况)。
- 我们使用了 `y < 0` 而实际上我们的意思是 `0 < y`。

作为测试人员, 我们一定要检查这些未处理的情况。请注意, 仅仅“修复错误”是不够的。如果我们不检查所有可能的情况, 错误还可能再次出现。作为测试人员, 我们希望能够系统地捕捉到所有可能的错误。这样简单的代码, 如果只是修正错误, 我们很可能在修正过程中再犯错, 不仅不能解决问题, 甚至还可能引入新的不同的错误。检查代码的目的并不是要找到错误(虽然这很重要), 而是要设计出能够捕获所有错误的测试集(或者现实一点, 捕获尽可能多的错误)。

需要注意的是: 循环有一个隐式的“`if`”, 它用于检测是否达到循环终止条件。因此, 循环也包含分支语句。当我们看到代码中的分支语句, 首先要考虑的问题是, “我们是否已经覆盖(测试)了所有分支?”令人惊讶的是, 对于实际代码, 覆盖所有分支并不总是可行的(因为在实际代码中, 根据需要, 一个函数可能被其他函数调用, 但调用并不是所有情况下都有必要进行的)。因此, 对于测试人员来说, 一个更一般的问题是, “你所要求的代码覆盖率是多少”? 答案最好是“我们测试大部分分支”, 然后解释为什么余下的分支很难测试。100% 的分支覆盖只是理想的情况。

26.3.4 系统测试

重要系统的测试是一项技术性工作。例如, 对电话系统的计算机控制子系统进行测试, 就需要在放置了许多机架式计算机的专门机房中进行, 通过模拟数万人的通话来测试控制系统。这种

测试系统本身的价值就达到数百万美元，而且需要非常熟练的工程师团队来实施测试。而电话系统一旦投入使用，其主要的电话交换机需要在持续运行 20 年的时间内最多停机 20 分钟（包括各种原因，如停电、水灾、地震等）。我们不会深入讨论这一问题，比起解决这个问题，教会一个物理系新生计算火星探测器的方向修正量会更容易些。但我们会给你介绍一些思想，这些思想对于测试一个较小的系统或者理解更大系统的测试会有所帮助。

首先，请记住测试的目的是发现错误，特别是那些可能频繁发生和很严重的错误。编写和运行大量的测试不是一件简单的工作。它要求测试人员要对待测系统有一定的理解。与单元测试相比，系统测试的有效性更依赖于应用程序的相关知识（领域知识）。开发一个系统不仅仅要用到编程语言和计算机科学知识，还需要对应用领域和用户的了解。这也是激励我们从事编程工作的重要原因之一：我们可以接触到许多有趣的应用程序，认识很多有趣的人。

一个完整的待测系统可能由许多组成部分（单元）构成，其构造可能会花费很长时间。因此，我们需要在开发过程中不断进行测试，一个可行的策略是：在完成所有单元测试之后，对于大量系统测试，每天只做一次（经常是在晚上开发人员睡觉的时候）。在这个过程中，回归测试是一项关键工作。最可能发生错误的地方是新加入的代码和以前发现过错误的代码。因此，重新运行旧的测试集（回归测试）是非常必要的。如果不这样做，一个大系统永远也不会达到稳定状态。因为在我们消除旧错误的同时，也可能引入新错误。

注意，当修正错误的时候，意外地引入一些新的错误是很正常的。但我们希望新错误的数量低于已排除的错误的数量，而且新错误的严重程度也低于老的错误。然而，至少在重新进行回归测试，并对新代码进行测试之前，我们必须假定系统是有问题的（我们的错误修正工作引起了问题）。

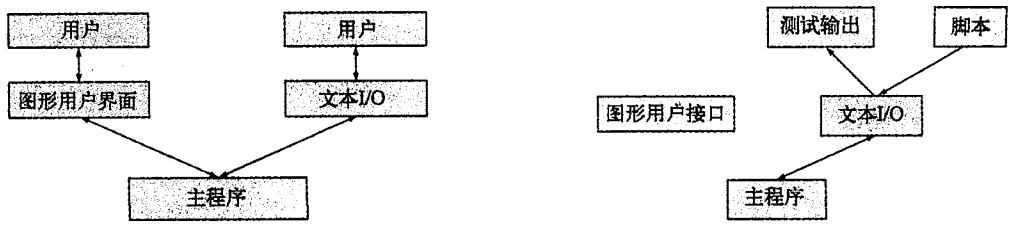
26.3.4.1 测试图形用户界面

设想你坐在屏幕前面，通过一个精致的图形用户界面系统地测试程序。我应该在哪儿点击鼠标？以什么顺序点击？我应该输入什么值？以什么顺序输入？对于一个大程序来说，想搞清楚这些问题是没什么希望的。因为可能性实在太多，或许我们可以考虑雇一群鸽子随机地在屏幕上乱啄（它们干这个工作是为了得到食物！）。雇用一群“普通的新手”，看着他们如何“乱啄”，其实并不罕见，而且还是很必要的，但这不是系统化的测试策略。任何实际的测试方案都要包含一些可重复的测试，这通常意味着与应用程序间的接口设计要绕过图形用户界面。

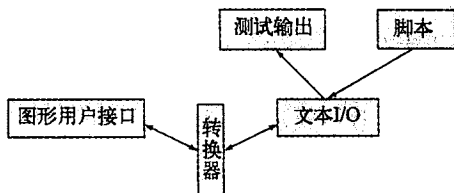
那么为什么还要让人坐在 GUI 程序前“乱啄”呢？原因很简单，用户可能会是孤僻的、笨拙的、幼稚的、世故的或急躁的人，测试人员不可能预见到用户的所有动作。即使系统已经经过最好的、最系统的测试，我们仍然需要真实的人来试用系统。经验表明，即使是最有经验的设计者、实现人员和测试人员，也可能预计不到实际系统用户的可能行为。有一个程序员的谚语，“当你建立起一个万无一失的系统的时候，大自然会创造一个更傻的傻瓜来破坏它”。

因此，理想情况下，GUI 只是调用一些定义明确的“主程序”的接口。也就是说，GUI 仅提供 I/O 操作，任何复杂的处理都与 I/O 分离。这表示我们可以提供不同的（非图形）接口，如左下图所示。

这样，在做单元测试的时候，我们可以为“主程序”编写或生成脚本，就像做单元测试那样（参见 26.3.2 节）。这样我们就可以将“主程序”的测试与 GUI 分离，如右下图所示。



有趣的是, 这样我们还可以半系统化地测试 GUI: 我们可以使用文本 I/O 来运行脚本, 然后通过 GUI 观察效果(假设我们将主程序的输出同时发送给 GUI 和文本 I/O 接口)。更进一步, 在测试 GUI 时, 我们可以绕过“主程序”: 提供一个文本命令, 能通过一个小的文本——GUI 命令“翻译器”直接发送给 GUI:



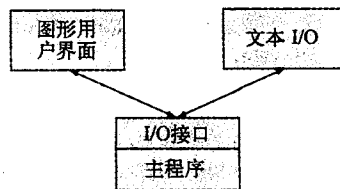
上述内容显示了关于“好的测试”的两个重要观点:

- 系统的各组成部分应该(尽可能地)能够独立测试。只有拥有清晰的接口定义的“单元”才能单独测试。
- 测试应该(尽可能地)可重复。人工参与的测试很难重复。

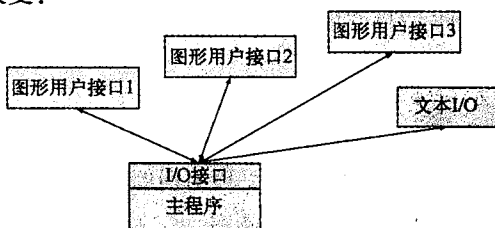
这也是我们所提及的“为测试而设计”的例子: 某些程序要比其他程序容易测试得多, 如果我们在设计的时候就考虑测试, 就可以构建出组织更为良好、更易测试的系统(参见 26.2 节)。组织更为良好是什么意思? 请看右图。这个图明显比前面的图要简单。在开始构造系统时, 不需要考虑太多, 在需要与用户通信的地方, 简单使用 GUI 库就可以了。这可能比我们预想的包括文本和图形 I/O 的程序代码量更少。那么, 如果我们的程序使用显式的用户界面, 并有更多的模块, 是否能比 GUI 代码到处散布的程序组织得更好呢?



为了实现两种用户界面, 我们需要仔细定义“主程序”与 I/O 间的接口。实际上, 我们需要定义一个通用的 I/O 接口层(类似于我们用来独立测试 GUI 的“翻译器”), 如下图所示。我们已经看过这方面的例子了: 第 13 ~ 16 章中的图形界面类。它们将“主程序”(即你写的代码)与“现成的”GUI 系统: FLTK、Windows、Linux 的 GUI 等分离。采用这种设计, 我们可以使用任何 I/O 系统。



这一点很重要吗? 我想的确是。首先, 它有助于测试, 没有系统化的测试就很难有严格的正确性。其次, 它带来可移植性。考虑如下场景: 你在一个小公司工作, 因为你喜欢苹果公司的电脑, 所以你的程序是在苹果计算机上开发的。现在, 你的公司逐渐发展起来了。你发现你的大部分潜在客户使用的是 Windows 或非 Mac 的 Linux 系统, 你该怎么办? 如果使用“简单”组织方式的话, (苹果的 Mac) GUI 命令会散布在你的代码的各个地方, 你必须重写所有这些代码。这是可以接受的, 因为程序可能隐藏有许多错误(这依赖于专门的测试), 重写时可以修正这些错误。但是, 你可以考虑另一种方法, GUI 和“主程序”相对分离(可以简化系统化测试)。这样, 当引入新的 GUI 界面时, 只需简单地将其连接到接口类(图中的“翻译器”)就可以了, 大部分代码都无需改变:



实际上,这种设计是一个使用显式的“瘦”接口显式分隔程序模块的典型例子。它与 12.4 节中“层”的概念类似。测试工作强烈要求程序能够清晰地划分为若干部分(每个部分都有相应的接口,我们可以用来进行测试)。

26.3.5 测试类

从技术上说,类的测试属于单元测试。但是,既然类包括了若干成员函数和声明,类的测试也可以看做系统测试。如果我们所测试的是基类,这一点更为明显,我们不得不考虑不同上下文(对应不同的派生类)。下面是 14.2 节中的 Shape 类:

```
class Shape { // deals with color and style, and holds sequence of lines
public:
    void draw() const; // deal with color and draw lines
    virtual void move(int dx, int dy); // move the shape +=dx and +=dy

    void set_color(Color col);
    Color color() const;

    void set_style(Line_style sty);
    Line_style style() const;

    void set_fill_color(Color col);
    Color fill_color() const;

    Point point(int i) const; // read-only access to points
    int number_of_points() const;

    virtual ~Shape() {}
protected:
    Shape();
    virtual void draw_lines() const; // draw the appropriate lines
    void add(Point p); // add p to points
    void set_point(int i, Point p); // points[i]=p;
private:
    vector<Point> points; // not used by all shapes
    Color lcolor; // color for lines and characters
    Line_style ls;
    Color fcolor; // fill color

    Shape(const Shape&); // prevent copying
    Shape& operator=(const Shape&);
};
```

我们应该怎么测试它呢?让我们先考虑一下 Shape 与 binary_search 的不同在哪里(从测试人员的角度出发):

- Shape 有若干函数。
- 一个 Shape 可以有多个状态(我们可以增加点、改变颜色等);这也意味着对一个函数的改变会影响到其他函数的行为。
- Shape 有虚函数,即 Shape 的行为依赖于其派生类(如果有的话)。
- Shape 不是一个算法。
- 对 Shape 的改变会显示在屏幕上。

最后一点非常糟糕。这意味着我们不得不让一个人始终盯着屏幕,查看 Shape 的行为是否像我们所希望的那样。这不利于实现系统的、可重复的、可负担的测试。正如 26.3.4.1 节中所说,我们要尽力避免这种情况的发生。不过,现在我们假定有一个警觉的检测者一直盯着屏幕,查看屏幕

上的显示是否偏离了我们的要求。

注意一个重要的细节：一个用户可以增加点，但是不能删除它们。用户或一个 Shape 可以读点的信息，但不能改变它们。从测试者的观点来看，任何不能改变(或至少假定不能改变)的东西都有助于我们的工作。

那么我们能测试什么，不能测试什么呢？为了测试 Shape，必须对其派生类进行独立和组合测试。但是，为了测试 Shape 对其特定派生类是否工作正常，还必须测试这个派生类，这大大增加了工作量。

我们注意到 Shape 的状态(值)主要是由 4 个成员定义的：

```
vector<Point> points;
Color lcolor;    // color for lines and characters
Line_style ls;
Color fcolor;    // fill color
```

对于一个 Shape，我们能做的就是尝试改变这些成员，然后观察效果。幸运的是，改变数据成员的唯一方式是通过成员函数定义的接口。

最简单的 Shape 是 Line，因此，我们(以最简单的方式)创建一个 Line，然后尝试所有可能的改变：

```
Line ln(Point(10,10), Point(100, 100));
ln.draw();           // see if it appears

// check the points:
if (ln.number_of_points() != 2) cerr << "wrong number of points";
if (ln.point(0)!=Point(10,10)) cerr<< "wrong point 1";
if (ln.point(1)!=Point(100,100)) cerr<< "wrong point 2";

for (int i=0; i<10; ++i) {    // see if it moves
    ln.move(i+5,i+5);
    ln.draw();
}

for (int i=0; i<10; ++i) {    // see if it moves back to where it started
    ln.move(i-5,i-5);
    ln.draw();
}
if (ln.point(0)!=Point(10,10)) cerr<< "wrong point 1 after move";
if (ln.point(1)!=Point(100,100)) cerr<< "wrong point 2 after move";

for (int i = 0; i<100; ++i) { // see if the color changes correctly
    ln.set_color(Color(i*100));

    if (ln.color() != i*100) cerr << "bad set_color";
    ln.draw();
}

for (int i = 0; i<100; ++i) { // see if the style changes correctly
    ln.set_style(Line_style(i*5));
    if (ln.style() != i*5) cerr << "bad set_style";
    ln.draw();
}
```

理论上，这个测试能够测试 Line 的创建、移动、改变颜色和风格等功能。实际上，我们需要更仔细地(和全面地)选择测试用例，就像我们在测试 `binary_search` 时所做的那样。而且，我们可以确定从文件中读取测试描述是一种更好的方案，这样我们也能获得一种更好的报告错误的方法。

此外，我们发现没有人能够跟上 Shape 的快速变化，这里有两个替代方案，我们可以

- 把程序的运行速度降下来，这样人们能够跟上 Shape 的变化。

- 找到 Shape 的一种描述形式，能通过程序来读取和分析。

我们几乎完全没有测试 `add(Point)`，对此，我们可以使用 `Open_polyline` 来进行测试。

26.3.6 寻找不成立的假设

`binary_search` 的规范明确要求输入序列必须是有序的。这个条件让我们不能利用许多单元测试的技巧。但这显然为编写糟糕代码提供了机会，我们已经设计的测试(系统测试除外)不能发现其中的错误。我们是否能利用对系统“单元”(函数、类等)的理解来设计更好的测试呢？

不幸的是，最简单的答案是否定的。作为纯粹的测试者，我们不能改变代码，而只能检查是否违反了接口的要求(前置条件)，我们应该在每次调用前进行前置条件检查，或者实现为函数的一部分(参见 5.5 节)。但是，只有待测代码是我们自己编写的，我们才可以插入这样的测试代码。如果我们是测试人员，并且代码的编写者会听从我们的要求(这并不总能实现)，我们可以告诉他们还未测试的内容，并要求他们确保这些内容被测试。

回到 `binary_search` 的例子：我们不能检测输入序列 `[first: last)` 是否是一个合法序列以及是否有序(参见 26.3.2.2 节)。但是，我们可以用下面的函数来检查：

```
template<class Iter, class T>
bool b2(Iter first, Iter last, const T& value)
{
    // check if [first:last) is a sequence:
    if (last < first) throw Bad_sequence();

    // check if the sequence is ordered:
    if (2 < last - first)
        for (Iter p = first + 1; p < last; ++p)
            if (*p < *(p - 1)) throw Not_ordered();

    // all's OK, call binary_search:
    return binary_search(first, last, value);
}
```

现在，`binary_search` 中没有包含这些测试代码，原因包括：

- 比较运算 `last < first` 不能用于前向迭代器；例如，`std::list` 的迭代器就没有 `<` (参见附录 B.3.2)。通常，没有一个真正好的办法来测试有一对迭代器定义了一个合法的序列(可以从 `first` 开始迭代，期待最终能遇到 `last`，但这不是一个好的检测方法)。
- 通过扫描整个序列来确定序列是否有序的代价远超过执行 `binary_search` 本身(`binary_search` 的目的不是盲目地遍历整个序列去查找某个值，这是 `std::find` 的做法)。

那我们能做什么呢？我们可以在测试中用 `b2` 来替代 `binary_search` (只在用随机访问迭代器调用 `binary_search` 时进行替换)。此外，如果允许的话，我们可以要求 `binary_search` 的编写者插入测试代码：

```
template<class Iter, class T> // warning: contains pseudo code
bool binary_search (Iter first, Iter last, const T& value)
{
    if (test enabled) {
        if (Iter is a random access iterator) {
            // check if [first:last) is a sequence:
            if (last < first) throw Bad_sequence();
        }

        // check if the sequence is ordered:
        if (first != last) {
            Iter prev = first;
            for (Iter p = ++first; p != last; ++p, ++prev)
                if (*p < *prev) throw Not_ordered();
        }
    }

    // all's OK, call binary_search:
    return binary_search(first, last, value);
}
```



```
    }  
}  
  
// now do binary_search  
}
```

其中,“test enabled”的含义依赖于测试是如何(为特定方式组织的专用系统)安排的,因此我们把它设为伪代码:在测试你自己的代码时,你就可以用一个 test_enabled 变量来代替。我们也把“Iter is a random access iterator”检测作为伪代码,因为我们没有解释“迭代器特征”(iterator traits)是什么。如果你真的需要做这个检测,你可以在高阶 C++ 书籍中查找迭代器特征的相关内容。

26.4 测试方案设计

在开始编写程序的时候,我们当然希望它最终是完整的、正确的。我们知道,为了达成这一目标,必须进行测试。因此,从编写程序的第一天起,我们在设计中就要考虑正确性和测试。实际上,许多优秀的程序员都有个口号“早测试,经常测试”,而且,他们不会在考虑好代码将来如何测试之前,就急于动手编写。及早考虑测试问题有助于避免发生在早期的错误(也有助于以后发现错误);我们赞成这种程序设计哲学。一些程序员甚至在编写程序单元之前就编写好了单元测试。

26.3.2.1 节和 26.3.3 节中的例子解释了这些关键的思想:

- 使用定义良好的接口,这样你可以测试这些接口的使用。
- 为各种操作定义文本描述方式,这样它们就可以被存储、分析和重放。这也包括输出操作。
- 在调用代码中嵌入对未检查假设(断言)的检测,以便在系统测试前捕获错误参数。
- 最小化依赖性,并且保持各种依赖关系清晰可见。
- 有一个清晰的资源管理策略。

从哲学上讲,这些思想可以看做是单元测试技术能很好地应用于子系统和整个系统的保证。

如果不考虑性能,我们可以始终进行未检查假设(要求、前置条件等)的检测。但是,程序通常不包含这部分代码,这是有原因的。例如,我们看到检查输入序列是否有序比 binary_search 本身还要复杂,并且代价更高。因此,设计一个系统能允许我们根据需要选择性地打开/关闭这种检测,是一个好想法。对大多数系统来说,在最终(发布)版本中留下相当多代价比较低的检查代码是个好主意:因为在一些“不可能”的情况发生的时候,我们更希望通过一个明确的错误信息来了解情况,而不是一个简单的系统崩溃。

26.5 调试

调试是一种技术,也是一种态度。显然,态度更重要。请回顾一下第 5 章,注意一下调试与测试的不同。这两者都捕获错误,但是调试更具专门性,重点关注排除已知错误和实现新特性。任何一种能让调试更像测试的方法我们都会去尝试。要说我们喜欢测试确实有些夸张,但是 we 确实讨厌调试。及早进行单元测试,在设计时考虑测试,都有助于最小化调试的工作量。

26.6 性能

对一个有用的程序来说,仅仅满足正确性是不够的。即使假定有足够的设备条件供程序使用,它也必须拥有一定的性能。一个好的程序应该是“效率足够高的”,即它能够在给定的资源条件下,在可接受的时间内得到结果。但要注意的是绝对的效率并不是我们的首要目标。一种固有

的认识是运行更快的程序可能会给开发工作带来麻烦,因为它会导致复杂的代码(可能包含更多的错误、调试工作量等大),使维护(包括移植和性能调优)工作更困难、代价更高。

那么,我们怎么才能知道一个程序(或程序单元)是“效率足够高的”呢?理论上讲,我们是不可能知道的。而且很多程序运行的硬件都非常快,使得这个问题不那么关键。我们曾经看到过这种情况:为了更好地检测系统发布后所发生的错误(即使是最好的代码,当它与其他代码一起工作时,错误也会发生),有的正式产品会以调试模式编译(虽然这可能导致运行速度比发布模式慢 25 倍)。

因此,对问题“效率是否足够高”的答案是:“测量感兴趣的测试用例花费多长时间”。在这里,你显然要充分了解最终用户“感兴趣”的是什麼,以及他们对于这些“感兴趣”的测试用例可接受的最长运行时间是多长。逻辑上,我们可以用秒表进行计时,并检查所花费的时间是否合理。在实际中,我们可以使用一些函数,例如 `clock()` (参见 26.6.1 节)来完成计时功能。而且,我们还可以自动比较测试所花费的时间与预先估计的合理时间。一种替代方法(或者与前一种比较同时做)是,记下测试花费的时间,与以前的测试进行比较。这是一种性能回归测试。

一些最糟糕的性能 bug 是由糟糕的算法引起的,这种问题也可以通过测试发现。使用大数据集进行测试的原因之一就是要暴露低效的算法。例如,假设有一个应用程序求矩阵每行元素之和(使用第 24 章中的 `Matrix` 库),下面是某人提供的相应函数:

```
double row_sum(Matrix<double,2> m, int n); // sum of elements in m[n]
```

现在,有人使用这个函数生成一个 `vector`, `v[n]` 保存前 `n` 行元素之和:

```
double row_accum(Matrix<double,2> m, int n) // sum of elements in m[0:n]
{
    double s = 0;
    for (int i=0; i<n; ++i) s+=row_sum(m,i);
    return s;
}

// compute accumulated sums of rows of m:
vector<double> v;
for (int i = 0; i<m.dim1(); ++i) v.push_back(row_accum(m,i+1));
```

设想这个函数是单元测试的一部分,或者是系统测试所测试的应用程序的一部分。不管在何种情况下,只要矩阵足够大,你都会发现一些奇怪的现象:基本上,程序所需要的时间与 `m` 的元素数目的平方成正比。为什么呢?因为函数先是求第一行所有元素之和,然后再求第二行所有元素之和(再次访问了第一行所有元素),接着求第三行所有元素之和(再次访问了前两行的所有元素),依此类推。

如果你认为这个例子不够好,那么想象一下如果 `row_sum()` 需要通过访问数据库读取数据的话,会发生什么吧。读硬盘会比读取主存慢几千倍。

现在,你可能会抱怨“没人会写出这么愚蠢的代码”!抱歉,我们见过更糟的。当隐藏在应用程序代码之中时,糟糕的算法(从性能的角度来看)是很难被发现的。当你第一次看这段代码的时候,你注意到性能问题了吗?除非你专门关注于这类问题,否则问题是很难被发现的。下面是一个来自真实的服务程序的例子:

```
for (int i=0; i<strlen(s); ++i) { /* do something with s[i] */ }
```

一般情况下, `s` 是一个包含 2 万个字符的字符串。

并不是所有的性能问题都是由糟糕的算法导致的。实际上(正如我们在 26.3.3 节中指出的),我们所编写的大部分代码不能归类为特定的算法。一般来说,这些“非算法”的性能问题都被归类为“糟糕的设计”。具体包括:

- 信息的重复计算(例如,上面的矩阵行求和问题)。

- 重复检查(例如,在循环中每次都检查数据的索引,或者在函数间传递参数时,即便参数没有改变,也重复检查它)。
- 重复访问硬盘(或网络)。

注意重复一词。显然,我们是指“不必要的重复”,但是,除非你重复很多次,这类操作不会对性能产生显著影响。对于函数参数和循环变量,我们当然要仔细检查。但是,如果对同一个值进行一百万次检查的话,这种冗余检查可能会伤害性能。如果通过测试,我们发现性能受到了影响,我们就要看看是否可以消除这些重复操作。然而,除非你认为性能是个关键问题,否则不要轻易删除代码。过早的优化反而可能浪费时间或引入更多错误。

26.6.1 计时

你怎么才能知道一段代码是否足够快呢?你如何确定一个操作的执行时间呢?在大部分情况下,你可以简单地通过看表来达到目的(秒表、挂钟或闹钟)。虽然这种方法不科学,也不准确,但是如果这种方法不可行的话(意味着你没有来得及看清花费了多少时间),通常你就可以下结论:程序足够快。但纠缠于性能问题并不是一个好的思路。

如果你希望获得精确时间,或者你不能坐在那里看秒表的话,你就需要计算机来帮助你。计算机可以获得准确的运行时间。例如,在 UNIX 系统中,只需要在所运行命令前加上 `time` 前缀,系统就会显示所花费的时间。你可以用 `time` 来查看编译一个 C++ 源文件 `x.cpp` 需要多长时间。通常,你的编译指令如下:

```
g++ x.cpp
```

如果要查看编译时间,可以加上 `time`:

```
time g++ x.cpp
```

上面的指令将编译 `x.cpp` 并将所花费的时间输出到屏幕上。对于小程序来说,这是一种简单、有效的办法。需要记住的是这种方法需要多运行几次,因为你的计算机上的“其他活动”可能影响计时的准确性。如果连续三次得到大致相同的结果,通常你就可以信任这一结果了。

但是,如果待测程序的运行时间是毫秒级的,应该怎么办呢?如果你希望更准确地测量程序的某一部分花费的时间,应该怎么办呢?你可以使用标准库函数 `clock()` 来计时,下面这个例子中就是采用这种方法测量函数 `do_something()` 所花费的时间:

```
#include <ctime>
#include <iostream>
using namespace std;

int main()
{
    int n = 10000000;           // repeat do_something() n times

    clock_t t1 = clock();       // start time
    if (t1 == clock_t(-1)) {     // clock_t(-1) means "clock() didn't work"
        cerr << "sorry, no clock\n";
        exit(1);
    }

    for (int i = 0; i < n; i++) do_something(); // timing loop

    clock_t t2 = clock();       // end time
    if (t2 == clock_t(-1)) {
        cerr << "sorry, clock overflow\n";
        exit(2);
    }
}
```

```

cout << "do_something() " << n << " times took "
    << double(t2-t1)/CLOCKS_PER_SEC << " seconds"
    << " (measurement granularity: "
    << CLOCKS_PER_SEC << " of a second)\n";
}

```

函数 `clock()` 返回值的类型为 `clock_t`。其中,在做除法前的显式类型转换 `double(t2 - t1)` 是必要的,因为, `clock_t` 可能是整数。何时用 `clock()` 开始计时依赖于具体程序,其目的是计量程序单次运行的起止时间间隔。`t1` 和 `t2` 是 `clock()` 的返回值, `double(t2 - t1)/CLOCKS_PER_SEC` 是以秒计量的两次调用之间的系统时间间隔。你可以在 `<ctime>` 中找到 `CLOCKS_PER_SEC` 的说明(“每秒 `clock` 计数”)。

如果处理器不支持 `clock()` 或者时间间隔太长的话, `clock()` 返回 `clock_t(-1)`。

函数 `clock()` 可以计量几分之一秒到几秒的时间间隔。例如,如果 `clock_t` 是 32 位有符号整数并且 `CLOCKS_PER_SEC` 是 1000000 的话,我们可以利用 `clock()` 以微妙为单位测量从 0 到超过 2000 秒(大约半小时)的时间间隔。

此外,对于任何一个测试对象,如果你不能连续三次得到大致相同的结果,则这个测试是不可信的。什么是“大致相同的结果”呢?合理的答案是“误差在 10% 以内”。现代计算机是非常快的:每秒执行 10 亿条指令是很普通的。这意味着很多程序的运行时间很难被测量,除非你把某个程序重复运行上万次或者它本身确实就很慢,例如有写硬盘或访问互联网的情况。对于后者,可能我们只需要运行重复几百次就可以了。但是对于如何正确理解这些实验结果,可能会有一些困难。

26.7 参考文献

Stone, Debbie, Caroline Jarrett, Mark Woodroffe 和 Shailey Minocha. *User Interface Design and Evaluation*. Morgan Kaufmann, 2005. ISBN 0120884364.

Whittaker, James A. *How to Break Software: A Practical Guide to Testing*. Addison-Wesley, 2003. ISBN 03211194330.



简单练习

运行下列 `binary_search` 程序的测试:

- 实现 26.3.2.2 节中 Test 的输入操作符。
- 对来自 26.3 节的序列,将测试描述补充完整,存入文件中:
 - `{1,2,3,5,8,13,21}` // an “ordinary sequence”
 - `{}`
 - `{1}`
 - `{1,2,3,4}` // even number of elements
 - `{1,2,3,4,5}` // odd number of elements
 - `{1,1,1,1,1,1,1}` // all elements equal
 - `{0,1,1,1,1,1,1,1,1,1,1}` // different element at beginning
 - `{0,0,0,0,0,0,0,0,0,0,0,1}` // different element at end
- 基于 26.3.1.3 节的内容,编写一个程序,它可以生成
 - 一个非常大的序列(你认为什么是大,为什么?)
 - 十个序列,每个序列的元素个数是随机的
 - 十个序列,每个序列分别包含 0,1,2...9 个随机数作为元素(但仍然有序)
- 重复上述测试,但序列中元素为字符串,例如 `{Bohr Darwin Einstein Lavoisier Newton Turing}`。



思考题

- 制作一个应用程序的列表,对每个程序都给出可能导致最严重后果的 bug 的简单说明。例如,航班控制

程序——坠机：231人死亡；损失价值5亿美元的设备。

2. 为什么我们不直接证明程序的正确性呢？
3. 单元测试与系统测试有什么不同？
4. 什么是回归测试，为什么它很重要？
5. 测试的目的是什么？
6. 为什么 `binary_search` 不检查它的要求？
7. 如果我们不能检查到所有错误，那么我们应该主要查找哪类错误？
8. 在处理数据序列时，错误最可能出现在代码的哪部分？
9. 为什么在测试时候使用大数值是个好主意？
10. 为什么测试用例经常以数据而不是代码形式出现？
11. 为什么我们要使用大量基于随机数的测试用例？什么时候使用？
12. 为什么测试带有 GUI 的程序很困难？
13. 为什么需要独立测试一个“单元”？
14. 可测试性和可移植性的联系是什么？
15. 为什么测试一个类要比测试一个函数困难？
16. 为什么测试的可重复性很重要？
17. 当发现一个“单元”依赖于未检查的假设(前置条件)时，测试者应该怎么办？
18. 程序设计者/实现者应该如何做，才能改进测试？
19. 测试与调试有什么不同？
20. 什么时候性能是我们要考虑的因素？
21. 对于如何(容易地)制造低性能问题，给出两个(或更多)例子。

术语

假设	后置条件	测试覆盖	黑盒测试
前置条件	测试工具	分支	证明
测试	<code>clock()</code>	回归	时间
为测试而设计	资源使用	单元测试	输入
状态	白盒测试	输出	系统测试

习题

1. 使用 26.3.2 节中的测试用例测试 26.1 节中的 `binary_search` 算法程序。
2. 修改 `binary_search` 的测试，使它能够处理任意数据类型，然后测试 `string` 序列和浮点序列。
3. 使用接受比较操作参数的 `binary_search` 版本，重复 26.3.2 节中的练习。列出引入额外的参数后，可能出现的新错误。
4. 设计一种测试数据的格式，可以让你只需定义一次数据序列，但可以在多个测试中使用。
5. 在 `binary_search` 的测试集中加入一个测试，它能够捕获 `binary_search` 修改数据序列这种(不太可能的)错误。
6. 对第7章的计算器程序做一些尽可能小的改动，使它能够从文件输入数据，并可以将输出存入文件中(或者使用你的操作系统的 I/O 重定向功能)。然后为它设计一套可行的综合测试。
7. 测试 20.6 节中的“简单文本编辑器”程序。
8. 为第 12~15 章中的图形界面库增加一个文本界面。例如，字符串“`Circle(Point(0,1),15)`”应该生成一个调用 `Circle(Point(0,1),15)`。使用这个文本界面生成一个“儿童图画”：一个带屋顶的二维房子，两个窗户和一个门。
9. 为图形界面库增加一个基于文本的输出格式。例如，当调用 `Circle(Point(0,1),15)` 时，一个字符串“`Circle(Point(0,1),15)`”也应被发送到输出流中。
10. 使用练习 9 中的基于文本的界面为图形界面库写一个更好的测试。

11. 测量 26.6 节中的矩阵求和例子的时间，其中矩阵是方阵，维度分别是 100、10 000、1 000 000 和 10 000 000。元素值是 $[-10:10]$ 之间的随机数。使用一个更有效的算法（非 $O(n^2)$ ）重写程序，并比较所花费的时间。
12. 写一个程序，它能生成随机浮点数并用 `std::sort()` 对这些数进行排序。比较 500 000 个和 5 000 000 个 `double` 值进行排序所花费的时间。
13. 重复上一个练习中的实验，但使用的是随机字符串，长度范围是 $[0:100)$ 。
14. 重复上一个练习，但是使用 `map` 而不是 `vector`，这样我们就不需要进行排序了。

附言

作为程序员，我们梦想能够编写出第一次运行就通过的完美程序。但是现实是残酷的：保证程序的正确性是很困难的，而且当我们（和我们的同事）改进代码时，很难使程序保持在正确的状态。测试，包括在设计时考虑测试问题，是保证我们提交的系统真正正常运行的主要方法。无论何时，当我们在这个高科技时代结束一天工作的时候，我们真的应该对（经常被忘记的）测试人员报以善意。

“C 是一种强类型、弱检查的程序设计语言”

——Dennis Ritchie

本章简要概述 C 语言及其标准库，我们假定读者已经掌握了 C++ 语言。我们列出 C 不支持的 C++ 特性，并通过例子程序展示应对这些缺失特性。我们还会讨论 C 和 C++ 不兼容的地方，以及 C 和 C++ 的互操作方式。我们会通过举例来说明 I/O、列表操作、内存管理和字符串操作方面的 C 特性。

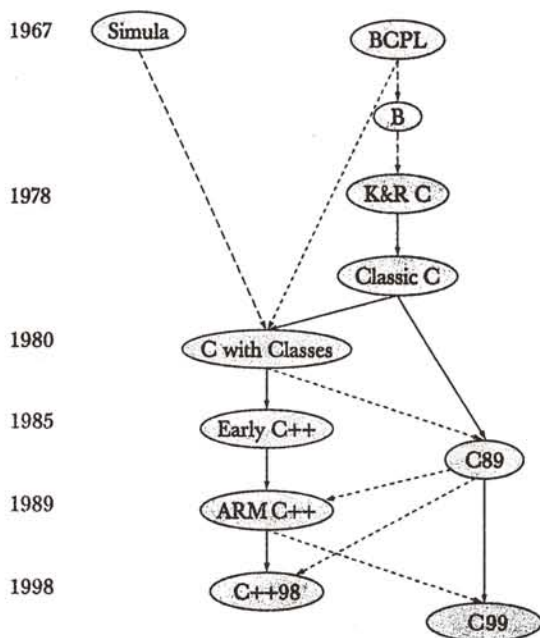
27.1 C 和 C++：兄弟

C 语言是由贝尔实验室的 Dennis Ritchie 设计和开发的，Brain Kernighan 和 Dennis Ritchie 合著的《The C Programming Language》一书（俗称“K&R”）使它迅速普及。这本书可能是迄今为止最好的 C 语言入门书籍和最好的程序设计书籍之一（参见 22.2.5 节）。C++ 最初的定义文档，是在 Dennis Ritchie 的 1980 年版 C 定义的基础上修改而来的。在此之后，两种语言都有了进一步的发展。与 C++ 一样，现在 C 语言的标准制定也是由 ISO 标准组织负责的。

我们大致上可以将 C 看做 C++ 的子集。因此，从 C++ 的角度看，介绍 C 语言就归结为两个问题：

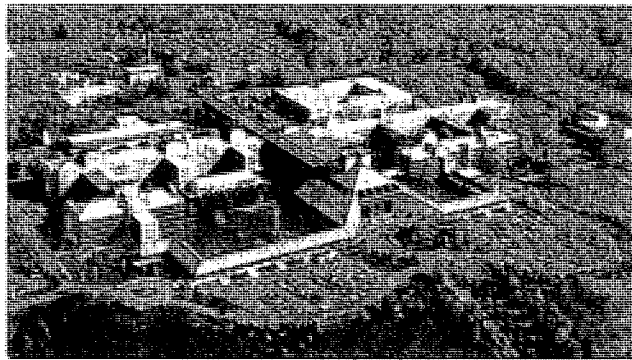
- C 的哪些特性并非 C++ 的子集。
- C++ 的哪些特性 C 并不支持，以及用什么样的 C 特性和技术可以弥补。

历史上，现代 C 语言和现代 C++ 语言是兄弟关系，两者都是“经典 C”的直系后裔。这里的“经典 C”是指 Brain Kernighan 和 Dennis Ritchie 的《The C Programming Language》第 1 版中介绍的 C 再加上结构赋值和枚举类型两个特性：



在所有的 C 版本中, C89 (K&R 第 2 版) 目前占据统治地位, 本章介绍的就是 C89。目前还有其他一些经典 C 在使用, C99 也有一些应用, 但只要你掌握了 C++ 和 C89, 使用这些不同的“方言”都不成问题。

C 和 C++ 都“出生”在新泽西州茉莉山贝尔实验室的计算机科学研究中心(有段时间, 我的办公室和 Brain Kernighan、Dennis Ritchie 的办公室就隔着一个走廊和几道门):



C 和 C++ 的标准制定目前都由 ISO 标准委员会负责。两种语言都有大量的实际实现在使用中。现在的编译系统通常都同时支持 C 和 C++, 通过编译选项或源程序后缀选择是按哪种语言编译。两种语言比其他任何语言都要更普及。两者最初的设计目的和当前最重要的应用都是系统级程序设计, 例如:

- 操作系统内核
- 设备驱动
- 嵌入式系统
- 编译器
- 通信系统

等价的 C 和 C++ 程序在性能上没有什么差异。

与 C++ 一样, C 的应用非常广泛。两者合并计算的话, 其开发社群应该是地球上最大的软件开发社群。

27.1.1 C/C++ 兼容性

我们经常会听到“C/C++”这种提法。但是, 并不存在这种语言, 这种提法是无知的表现。我们只在讨论 C/C++ 兼容性问题, 以及论及大的 C/C++ 共享技术社群时才会用“C/C++”的说法。

C++ 大体上是(但不完全是)C 的一个超集。大部分 C 和 C++ 都有的特性, 其语义在两种语言中也是相同的, 例外情况很少。C++ 的设计目标之一就是“尽可能接近 C, 直到不能再近”, 目的在于:

- 易于两种语言间的转换
- 两种语言的共存

两者的不兼容之处大多与 C++ 更严格的类型检查有关。

下面是一个合法的 C 语言程序, 但它不是合法的 C++ 程序, 原因在于其中一个标识符(class)是 C++ 的关键词, 但不是 C 的关键词:

```
int class(int new, int bool); /* C, but not C++ */
```

下面是一个在两种语言中都合法, 但语义不同的例子:


```
int s = sizeof('a');          /* sizeof(int), often 4 in C and 1 in C++ */
```

在 C 语言中, 字符常量(如 'a') 的类型是 int, 而在 C++ 中是 char。但是, 对于一个 char 型变量 ch, 两种语言中都有 sizeof(ch) == 1。

关于兼容性和语言差异的话题总是不那么令人兴奋, 因为里面没有什么新的程序设计技术可学。你可能会对 printf() (参见 27.6 节) 感兴趣, 但除此之外(以及一些无用的工程师间的幽默) 本章显得有些干巴巴的。本章的目的很简单: 使你能读写 C 程序(如果你有这种需求的话)。本章内容主要是指出一些潜在的危险: 一些有经验的 C 程序员认为很显然, 但对于 C++ 程序员通常很意外的东西。我们希望你以最小的代价学会规避这些危险, 而不必在实际应用中撞得头破血流。

大多数 C++ 程序员迟早都会遇到必须处理 C 代码的情况, 就像大多数 C 程序员必须处理 C++ 代码一样。本章介绍的很多内容对于大部分 C 程序员来说都是很熟悉的内容, 但也有一些内容属于“专家级知识”。原因很简单: 人们对什么是“专家级”很难达成共识, 我们只是介绍那些实际程序中很常见的问题。也许理解兼容性问题是赢得“C 专家”赞誉的捷径, 但请记住: 真正的专家水平是指使用语言(本章中是 C)的水平高超, 而不是指理解了一些深奥的语言规则(如一些兼容性问题)。

参考文献

- ISO/IEC 9899:1999. *Programming Languages – C*. This defines C99; most implementations implement C89 (often with a few extensions).
- ISO/IEC 14882:2003-04-01 (second edition). *Programming Languages – C++*. From a programmer's point of view, this standard is identical to the 1997 version.
- Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. ISBN 01311036 28.
- Stroustrup, Bjarne. “Learning Standard C++ as a New Language.” *C/C++ Users Journal*, May 1999.
- Stroustrup, Bjarne. “C and C++: Siblings”; “C and C++: A Case for Compatibility”; and “C and C++: Case Studies in Compatibility.” *The C/C++ Users Journal*, July, Aug., and Sept. 2002.

我的论文在我的主页上很容易找到。

27.1.2 C 不支持的 C++ 特性

从 C++ 的角度, C (即 C89) 缺少很多特性, 例如:

- 类和成员函数
 - 解决方法: 使用 struct 和全局函数。
- 派生类和虚函数
 - 解决方法: 使用 struct、全局函数和函数指针(参见 27.2.3 节)
- 模板和内联函数
 - 解决方法: 使用宏(参见 27.8 节)。
- 异常
 - 解决方法: 使用错误代码、错误返回值等。
- 函数重载
 - 解决方法: 不同函数使用不同名字。
- new/delete:
 - 解决方法: 使用 malloc()/free() 和分离的初始化/结束处理代码。
- 引用

- 解决方法: 使用指针。
- 常量表达式中的 `const`
 - 解决方法: 使用宏。
- `for` 语句中的声明和作为语句的声明
 - 解决方法: 将所有声明都放在语句块的头部, 或者为每组定义引入一个新的语句块。
- `bool` 类型
 - 解决方法: 使用 `int`。
- `static_cast`、`reinterpret_cast` 和 `const_cast`
 - 解决方法: 使用 C 风格的类型转换(如 `(int)a`)来替代 C++ 风格的 `static <int>(a)`。
- `//` 注释
 - 解决方法: 使用 `/* ... */` 注释。

很多有用的代码都是用 C 写的, 因此上面这个列表实际上在提醒我们: 没有什么语言特性是绝对必要的。很多语言特性, 甚至是大多数 C 语言特性, 只是为了方便程序员编写程序而设计的。毕竟, 如果你足够聪明、足够有耐心, 而且给你足够时间的话, 任何程序都可以用汇编语言写出来。注意, 由于 C 和 C++ 都使用相同的机器模型, 而且这个模型非常接近实际计算机, 因此它们都非常适合于模拟很多不同的程序设计风格。

本章剩余部分将介绍在没有这些特性的情况下如何来编写有用的程序。对于使用 C 语言, 我们的基本建议如下:

- 用 C 语言特性来模拟 C++ 特性所支持的程序设计技术。
- 当编写 C 程序时, 使用 C++ 中的 C 子集。
- 调整编译器的警告级别, 确保进行函数参数检查。
- 对于大型程序, 使用 `lint`(参见 27.2.2 节)。

很多 C/C++ 不兼容之处的细节相当晦涩难懂。但是, 如果只是读写 C 程序, 大多数细节你都不必记忆, 因为:

- 当你使用 C 不支持的 C++ 特性时, 编译器会提醒你。
- 如果遵循上述原则, 你几乎不会遇到相同语句在 C 和 C++ 中语义不同的情况。

虽然缺少上述 C++ 特性, 但也有一些特性在 C 中更为重要:

- 数组和指针
- 宏
- `typedef`
- `sizeof`
- 类型转换

本章中也会给出这些特性的一些例子。

我将 C 的前身 BCPL 中的 `//` 注释引入了 C++, 因为我真的厌烦了输入 `/* ... */` 方式的注释。很多 C 的方言包括 C99 都支持 `//`, 因此它在很多情况下是安全的, 尽管使用就是。但在本章中, 对于 C 语言的例子程序, 我们只使用 `/* ... */` 注释。C99 吸纳了一些 C++ 的特性(以及一些和 C++ 兼容的特性), 但在本章中我们使用 C89, 因为 C89 要普及得多。

27.1.3 C 标准库

很自然, C++ 中与类和模板相关的特性 C 是不支持的, 包括:

- vector
- map
- set
- string
- STL 算法: 如 `sort()`、`find()` 和 `copy()`
- `iostream`
- `regex`

对于这些特性, 我们通常可以用基于数组、指针和函数的 C 标准库特性来完成类似功能。C 标准库主要包括如下部分:

- `<stdlib.h>`: 一般工具(如 `malloc()` 和 `free()`, 参见 27.4 节)。
- `<stdio.h>`: 标准 I/O, 参见 27.6 节。
- `<string.h>`: C 风格字符串处理和内存管理, 参见 27.5 节。
- `<math.h>`: 标准浮点算术函数, 参见 24.8 节。
- `<errno.h>`: `<math.h>` 所涉及的错误码, 参见 24.8 节。
- `<limits.h>`: 整数类型的大小, 参见 24.2 节。
- `<time.h>`: 日期和时间, 参见 26.6.1 节。
- `<assert.h>`: 调试用的断言, 参见 27.9 节。
- `<ctype.h>`: 字符集, 参见 11.6 节。
- `<stdbool.h>`: 布尔宏。

这些标准库特性的完整说明, 请查阅好的 C 语言教材, 如 K&R。所有这些库(和头文件)也存在于 C++ 中。

27.2 函数

在 C 中:

- 函数不能重名。
- 函数参数类型检查是可选的, 不是强制的。
- 没有引用类型(因而参数传递也没有传引用方式)。
- 没有成员函数。
- 没有内联函数(C99 除外)。
- 有可选的函数定义语法。

除此之外, C 的函数与 C++ 很相似。我们来逐个考察这些差别。

27.2.1 不支持函数名重载

考虑下面代码:

```
void print(int);           /* print an int */
void print(const char*);   /* print a string */ /* error! */
```

第二个声明是错误的, 因为两个函数不能同名。所以你必须设计两个恰当的名字:

```
void print_int(int);       /* print an int */
void print_string(const char*); /* print a string */
```

不支持函数重载有时还会被认为是一个好的特性: 不会发生使用错误的函数输出整数的意外情况了! 显然, 我们并不接受这种说法, 不支持函数重载使得泛型程序设计思想变得很尴尬, 因为泛型程序设计的重要特点就是用相同的名字表示语义相似的函数。

27.2.2 函数参数类型检查

考虑下面代码：

```
int main()
{
    f(2);
}
```

它在 C 编译器中会顺利编译通过，也就是说，你不必在使用函数之前声明函数（虽然你可以这么做，而且也应该这么做）。f() 可能定义在程序某个地方，也可能在其他文件中，但如果并未定义的话，连接程序就会报错。

不幸的是，即使 f() 定义在其他文件中，它也有可能是这样的：

```
/* other_file.c: */

int f(char* p)
{
    int r = 0;
    while (*p++) r++;
    return r;
}
```

连接程序不会报告这个错误。你只会得到一个运行时错误，或者是奇怪的运行结果。

我们如何应对这类问题呢？头文件的一致使用是一个实用的方法。如果你调用或定义的每个函数都在一个特定的头文件中声明，而且无论是否用到，每个程序文件都包含此头文件，那么我们就可以确保函数声明检查。然而，在大型程序中，这很难做到。因此，大多数 C 编译器都具备对应的编译选项，来选择是否对未定义函数的调用给出警告，我们可以利用这一特性。而且，从 C 语言出现的早期开始，就已经有了能检查代码一致性问题程序。这些程序通常称为 lint。在编译每个大型程序之前，我们都应该使用 lint 来检查程序。你会发现 lint 会推动你以非常类似 C++ 子集的方式来使用 C。因为 C++ 最初的一个设计目标就是使编译器能容易地检查更多（而不是所有）lint 能检查的问题。

你可以让 C 进行函数参数检查。这很简单，只要在函数声明时指出参数类型即可（如同 C++ 那样）。这种函数声明称为函数原型（function prototype）。但是，要小心那些未指定参数的函数声明，那些声明不是函数原型，并不保证进行函数参数检查：

```
int g(double);    /* prototype — like C++ function declaration */
int h();          /* not a prototype — the argument types are unspecified */

void my_fct()
{
    g();           /* error: missing argument */
    g("asdf");    /* error: bad argument type */
    g(2);          /* OK: 2 is converted to 2.0 */
    g(2,3);        /* error: one argument too many */

    h();           /* OK by the compiler! May give unexpected results */
    h("asdf");    /* OK by the compiler! May give unexpected results */
    h(2);          /* OK by the compiler! May give unexpected results */
    h(2,3);        /* OK by the compiler! May give unexpected results */
}
```

其中，h() 的声明没有指定参数类型。这并不意味着 h() 不接受参数，而是意味着“接受任何参数集合，期望它们与被调函数匹配”。再次申明，好的编译器会对这类问题给出警告，而 lint 可以检查出这类问题。

C++	C 等价语法
<code>void f();</code> // 首选方式	<code>void f(void);</code>
<code>void f(void);</code>	<code>void f(void);</code>
<code>void f(...);</code> // 接受任何参数	<code>void f();</code> /* 接受任何参数 */

对于作用域中找不到函数原型的情况，C 定义了一组特殊的规则来转换参数。例如，char 和 short 会转换为 int，而 float 会转换为 double。如果需要了解相关内容，比如说 long 如何处理，请查阅好的 C 教材。我们的建议很简单：所有函数都应该给出函数原型。

注意，即使错误类型的参数通过了编译器的检查，例如将一个 char * 传递给了 int 型的参数，在其使用中也会出错。如 Dennis Ritchie 所说：“C 是一个强类型、弱检查的程序设计语言。”

27.2.3 函数定义

你可以像在 C++ 中一样定义函数，定义本身就是函数原型：

```
double square(double d)
{
    return d*d;
}

void ff()
{
    double x = square(2);      /* OK: convert 2 to 2.0 and call */
    double y = square();      /* argument missing */
    double y = square("Hello"); /* error: wrong argument type */
    double y = square(2,3);    /* error: too many arguments */
}
```

没有参数的函数定义不是函数原型：

```
void f() { /* do something */ }

void g()
{
    f(2); /* OK in C; error in C++ */
}
```

其中

```
void f(); /* no argument type specified */
```

意为“f() 可以接受任意数目、任意类型的参数”，这看起来真的很奇怪。为此，我发明了一种新的语法，用关键字 void 显式表示“什么都没有”的含义（英文单词 void 的意思就是“什么都没有”）：

```
void f(void); /* no arguments accepted */
```

我很快就后悔了，因为这种方式看起来很奇怪，而且如果能够保证进行参数类型检查的话，这种方式完全是多余的。更糟的是，Dennis Ritchie (C 语言之父) 和 Doug McIlroy (在贝尔实验室计算机科学研究中心内部，他是审美方面的最终裁决者) 都称 void 参数是“讨厌的”。不幸的是，这个讨厌的东西在 C 社群内已经非常流行了。在 C++ 中不要使用它，因为在 C++ 中，它不仅丑陋，而且逻辑上是多余的。

C 还提供另一种 Algol60 风格的函数定义方式——参数类型 (可选的) 与参数名分离：

```
int old_style(p,b,x) char* p; char b;
{
    /* ... */
}
```

这种“旧式定义”比 C++ 的历史还早，它也不是函数原型。默认情况下，未指定类型的参数被当做 int 型。因此，x 是函数 old_style() 的一个整型参数。我们可以这样调用 old_style()：

```
old_style();           /* OK: all arguments missing */
old_style("hello", 'a', 17); /* OK: all arguments are of the right type */
old_style(12, 13, 14);    /* OK: 12 is the wrong type, */
                          /* but maybe old_style() won't use p */
```

编译器应该会接受这些调用(但我们期望它对第一个和第三个调用能给出警告)。

对于函数参数检查问题,我们的建议是:

- 坚持使用函数原型(使用头文件)。
- 设置编译器的警告级别,使它能捕获参数类型错误。
- 使用 lint。

遵循这些原则的结果就是,写出的 C 代码也是正确的 C++ 代码。

27.2.4 C++ 调用 C 和 C 调用 C++

如果编译器支持的话,你可以将 C 编译器编译出的目标文件和 C++ 编译出的目标文件连接在一起。例如,你可以将 GNU C/C++ 编译器(GCC)编译出 C 和 C++ 目标文件连接在一起。微软 C/C++ 编译器(MSC++)生成的 C 和 C++ 目标文件也可以连接在一起。这种开发方式很常见也很有用,你可以使用的库的范围大大拓宽,不必受限于一语言的库。

C++ 的类型检查比 C 严格。特别是, C++ 的编译器和连接器会检查两个函数 $f(int)$ 和 $f(double)$ 的定义和使用是否一致,即使定义和使用在不同的源文件中。而 C 连接器不会做这种检查。为了从 C++ 中调用 C 函数,以及从 C 中调用 C++ 函数,应该通知编译器我们的意图:

```
// calling C function from C++:
```

```
extern "C" double sqrt(double); // link as a C function

void my_c_plus_plus_fct()
{
    double sr = sqrt(2);
}
```

extern "C" 告知编译器使用 C 连接器约定。除此之外,与普通 C++ 函数没有什么区别。实际上, C++ 标准库函数 $\sqrt{\text{double}}$ 通常就是 C 标准库中的 $\sqrt{\text{double}}$ 。采用这种方法,无需对 C 程序进行任何特殊处理,其中的函数就能被 C++ 程序调用。我们要做的全部事情只是让 C++ 采用 C 的连接约定。

extern "C" 也可用来使 C++ 函数能被 C 调用:

```
// C++ function callable from C:
```

```
extern "C" int call_f(S* p, int i)
{
    return p->f(i);
}
```

这样,我们就可以在 C 程序中间调用成员函数 $f()$ 了:

```
/* call C++ function from C: */
```

```
int call_f(S* p, int i);
struct S* make_S(int, const char*);

void my_c_fct(int i)
{
    /* ... */
    struct S* p = make_S(x, "foo");
    int x = call_f(p, i);
    /* ... */
}
```

在 C 中无需(或不可能)声明这是一个 C++ 函数。

这种互操作性的好处是显而易见的:可以混合使用 C 和 C++ 编写程序。特别是, C++ 程序可以使用 C 库,而 C 程序也可以使用 C++ 库。

在上例中,我们假定 C 和 C++ 可以共享 p 指向的类对象。对于大多数类对象来说,这是没有问题的。特别是,如果你定义了下面这样的类:

```
// in C++:
class complex {
    double re, im;
public:
    // all the usual operations
};
```

你可以在 C++ 程序和 C 程序之间传递对象指针,甚至可以在 C 程序中访问 re 和 im,只需定义如下结构类型:

```
/* in C: */
struct complex {
    double re, im;
    /* no operations */
};
```

任何程序设计语言中的内存布局规则都可能很复杂,不同语言混合编程中的内存布局规则就更加难以说清。但对于 C 和 C++ 来说,内置类型和没有虚函数的类(struct)对象可以直接传递。如果类具有虚函数,你只能传递对象的指针,而且实际的处理工作应该交给 C++ 代码。call_f()就是一个例子:f()可能是虚函数,这个例子展示了如何在 C 程序中调用虚函数。

除了内置类型外,最简单也最安全的类型共享方式就是在一个公共头文件中定义 struct。但是,这种方法严重限制了 C++ 的编程模式,因此我们不会局限于这种方式。

27.2.5 函数指针

如果希望在 C 中使用面向对象技术(参见 14.2 ~ 14.4 节),应该怎么做呢?最重要的是要找到虚函数的替代技术。大多数人会马上想到一个主意:在 struct 中添加一个“类型域”来指明对象是基类还是某个派生类,例如,对于 shape 及其派生类,指明给定对象是哪种形状。如下面程序所示:

```
struct Shape1 {
    enum Kind { circle, rectangle } kind;
    /* ... */
};

void draw(struct Shape1* p)
{
    switch (p->kind) {
        case circle:
            /* draw as circle */
            break;
        case rectangle:
            /* draw as rectangle */
            break;
    }
}

int f(struct Shape1* pp)
{
    draw(pp);
    /* ... */
}
```

这段程序可以正常工作，但存在两处隐患：

- 我们必须为每个“伪虚函数”（如 `draw()`）编写一个新的 `switch` 语句。
- 每当加入一个新的形状，我们就必须修改每个“伪虚函数”（如 `draw()`），为 `switch` 语句增加一个 `case` 分支。

第二个问题非常讨厌，它意味着我们不可能将“伪虚函数”放在库中，因为用户不得不频繁修改这些函数。虚函数最有效的替代技术之一是函数指针：

```
typedef void (*Pfct0)(struct Shape2*);
typedef void (*Pfct1int)(struct Shape2*,int);

struct Shape2 {
    Pfct0 draw;
    Pfct1int rotate;
    /* ... */
};

void draw(struct Shape2* p)
{
    (p->draw)(p);
}

void rotate(struct Shape2* p, int d)
{
    (p->rotate)(p,d);
}
```

`Shape2` 的使用与 `Shape1` 一样：

```
int f(struct Shape2* pp)
{
    draw(pp);
    /* ... */
}
```

稍微做一点改动，对象就不必携带每个伪虚函数的指针，而是保存一个指向函数指针数组（对于 C++ 中虚函数的所有实现）的指针即可。在实际程序设计中，这种方法的主要问题是要正确初始化所有函数指针。

27.3 小的语言差异

本节介绍 C 和 C++ 之间的一些小的差异，如果你从未听说过这些差异的话，就容易犯错。一些差异还会严重影响与之明显相关的程序设计工作。

27.3.1 结构标签名字空间

在 C 中，`struct` 的名字（C 中没有类）与其他标识符位于不同的名字空间。因此，每个 `struct` 的名字（称为结构标签，`structure tag`）必须以关键字 `struct` 为前缀。例如：

```
struct pair { int x,y; };
pair p1;          /* error: no identifier "pair" in scope */
struct pair p2;    /* OK */
int pair = 7;      /* OK: the struct tag pair is not in scope */
struct pair p3;    /* OK: the struct tag pair is not hidden by the int */
pair = 8;          /* OK: "pair" refers to the int */
```

令人惊讶的是，归功于一个不很正当的兼容性漏洞，这段代码在 C++ 中也是正确的。变量（或者函数）与 `struct` 同名是一种常见的 C 语言用法，但我们并不推荐这样做。

如果你不希望在每个结构名之前写上 `struct` 前缀，可以使用 `typedef`（参见 20.5 节）。下面的程序写法很常见：


```
typedef struct { int x,y; } pair;
pair p1 = { 1, 2};
```

一般而言, typedef 在 C 中更为常见, 也更为有用, 因为在 C 语言中你没有办法定义附带操作的新类型。

在 C 中, 嵌套的 struct 的名字与包含它的 struct 位于同一个作用域中, 例如:

```
struct S {
    struct T { /* ... */ };
    /* ... */
};

struct T x; /* OK in C (not in C++) */
```

而在 C++ 中, 你必须这样写:

```
S::T x; // OK in C++ (not in C)
```

只要可能, 不要使用嵌入的 struct; 其作用域规则与大多数人的自然的(也是合理的)想法不同。

27.3.2 关键字

很多 C++ 关键字不是 C 关键字(因为 C 不支持对应的功能), 因此可以用作 C 标识符:

C++ 关键字而不是 C 关键字

and	and_eq	asm	bitand	bitor	bool
catch	class	compl	const_cast	delete	dynamic_cast
explicit	export	false	friend	inline	mutable
namespace	new	not	not_eq	operator	or
or_eq	private	protected	public	reinterpret_cast	static_cast
template	this	throw	true	try	typeid
typename	using	virtual	wchar_t	xor	xor_eq

不要将这些名字用作 C 标识符, 否则你的程序将无法移植为 C++ 程序。如果你在头文件中使用了一些名字, 头文件将无法被 C++ 使用。

一些 C++ 关键字是 C 中的宏:

C++ 关键字, C 中的宏

and	and_eq	bitand	bitor	bool	compl	false	
not	not_eq	or	or_eq	true	wchar_t	xor	xor_eq

在 C 中, 这些宏是在 <iso646.h> 和 <stdbool.h> (bool、true、false) 中定义的。不要利用它们是 C 的宏这一特性。

27.3.3 定义

与 C 相比, C++ 允许将定义放置在程序更多的地方。例如:

```
for (int i = 0; i < max; ++i) x[i] = y[i]; // definition of i not allowed in C

while (struct S* p = next(q)) {          // definition of p not allowed in C
    /* ... */
}

void f(int i)
{
    if (i < 0 || max <= i) error("range error");
    int a[max]; // error: declaration after statement not allowed in C
    /* ... */
}
```

C(C89)不允许在 for 语句的初始化部分、条件判断或者语句块中语句之后放置声明。这段代码在 C 中必须这样写：

```
int i;
for (i = 0; i < max; ++i) x[i] = y[i];

struct S* p;
while (p = next(q)) {
    /* ... */
}

void f(int i)
{
    if (i < 0 || max <= i) error("range error");
    {
        int a[max];
        /* ... */
    }
}
```

在 C++ 中，未初始化的声明被视为一个定义，但在 C 中，仍被视为一个声明，因此可以重复多次：

```
int x;
int x; /* defines or declares a single integer called x in C; error in C++ */
```

在 C++ 中，每个实体只能定义一次。这引出一个有趣的问题：如果不同源文件中有两个同样的定义，会发生什么情况？

```
/* in file x.c: */
int x;

/* in file y.c: */
int x;
```

单独编译 x.c 或 y.c 的话，C 和 C++ 编译器都不会发现错误。但是，如果在 C++ 中将 x.c 和 y.c 一起编译，连接程序会报告“重复定义”错误。如果两者是在 C 中一起编译，连接会顺利通过，因为（根据 C 的规则）连接程序认为 x.c 和 y.c 共享一个 x。但最好不要用这种方式，如果你确实希望不同源文件共享一个全局变量 x，应该显式地声明：

```
/* in file x.c: */
int x = 0; /* the definition */

/* in file y.c: */
extern int x; /* a declaration, not a definition */
```

更好的方式是使用头文件：

```
/* in file x.h: */
extern int x; /* a declaration, not a definition */

/* in file x.c: */
#include "x.h"
int x = 0; /* the definition */

/* in file y.c: */
#include "x.h"
/* the declaration of x is in the header */
```

当然，最好的方式是避免使用全局变量。

27.3.4 C 风格类型转换

在 C 中(C++ 中也可以)，你可以用下面这样的简单语法来实现值 v 向类型 T 的转换：

(T)v

这种“C 风格类型转换”或称为“老式类型转换”，受到打字不熟练或者马虎的程序员们的欢迎，因为它非常简单，而且你不必了解 v 到底是如何转换为类型 T 的。但代码维护人员却很害怕这种形式，因为它几乎是隐形的，而且对于程序作者的意图没有给出任何线索。C++ 风格的类型转换（或称为新式类型转换或模板方式转换参见附录 A.5.7）是显式的类型转换，因此易于发现、意义明确。但在 C 中，我们只能使用老式的类型转换：

```
int* p = (int*)7; /* reinterpret bit pattern: reinterpret_cast<int*>(7) */
int x = (int)7.5; /* truncate double: static_cast<int>(7.5) */

typedef struct S1 { /* ... */ } S1;
typedef struct S2 { /* ... */ } S2;
S2 a;
const S2 b; /* uninitialized consts are allowed in C */

S1* p = (S1*)&a; /* reinterpret bit pattern: reinterpret_cast<S1*>(&a) */
S2* q = (S2*)&b; /* cast away const: const_cast<S2*>(&b) */
S1* r = (S1*)&b; /* remove const and change type; probably a bug */
```

即使在 C 语言中，我们也很犹豫是否推荐使用宏（参见 27.8 节），但宏又确实可以表达某些思想：

```
#define REINTERPRET_CAST(T,v) ((T)(v))
#define CONST_CAST(T,v) ((T)(v))

S1* p = REINTERPRET_CAST (S1*,&a);
S2* q = CONST_CAST(S2*,&b);
```

这种方法并不具备 `reinterpret_cast` 和 `const_cast` 的类型检查能力，而且很丑陋，但它至少易于发现，也使程序员的意图更为明显。

27.3.5 void* 的转换

在 C 中，我们可以将 `void*` 赋予任何指针类型的变量，或用它来初始化指针变量，在 C++ 中这是不可以的。例如：

```
void* alloc(size_t x); /* allocate x bytes */

void f(int n)
{
    int* p = alloc(n*sizeof(int)); /* OK in C; error in C++ */
    /* ... */
}
```

在这段代码中，`alloc()` 返回的 `void*` 值被隐式地转换为 `int*` 类型。而在 C++ 中，我们必须这样写：

```
int* p = (int*)alloc(n*sizeof(int)); /* OK in C and C++ */
```

这条语句使用了 C 风格的类型转换（参见 27.3.4 节），因此在 C 和 C++ 中都是正确的。

为什么在 C++ 中 `void*` 到 `T*` 的隐式类型转换是非法的呢？因为这种转换可能是不安全的：

```
void f()
{
    char i = 0;
    char j = 0;
    char* p = &i;
    void* q = p;
    int* pp = q; /* unsafe; legal in C, error in C++ */
    *pp = -1; /* overwrite memory starting at &i */
}
```

在这段代码中，我们甚至不能确定哪个内存区域被更改了。也许是 j 和 p 的一部分？也许是用于 f()

的调用管理的内存区域(f的调用栈)? 无论如何, 对 f() 的调用都是一件糟糕的事情。

注意, 从 T* 到 void* 的(反向)转换是百分之百安全的, 这样的转换不会形成像上面代码一样糟糕的程序, 因此在 C 和 C++ 中都允许这样的转换。

不幸的是, 隐式的 void* 到 T* 的类型转换在 C 程序中随处可见, 这也许是实际程序中最主要的 C/C++ 兼容性问题。

27.3.6 枚举

在 C 中, 我们可以将一个 int 值赋予一个 enum 变量, 而无需类型转换。例如:

```
enum color { red, blue, green };
int x = green;           /* OK in C and C++ */
enum color col = 7;      /* OK in C; error in C++ */
```

这个特性意味着, 在 C 中我们可以对枚举变量进行增 1(++) 和减 1(--) 运算。这可能很方便, 但会导致灾难性的后果:

```
enum color x = blue;
++x;      /* x becomes green; error in C++ */
++x;      /* x becomes 3; error in C++ */
```

这段代码中, x“跌出”了枚举常量的范围, 可能我们就是想这么做, 但也很有可能是我们无意中犯的错误。

注意, 与结构标签类似, 枚举类型名也位于自己独立的名字空间中, 因此使用时必须使用关键字 enum 作为前缀:

```
color c2 = blue;      /* error in C: color not in scope; OK in C++ */
enum color c3 = red;  /* OK */
```

27.3.7 名字空间

C 不支持名字空间(这里的“名字空间”一词是指在 C++ 中的含义)。那么在大型 C 程序中应该如何避免名字冲突呢? 一般的策略是使用前缀和后缀。例如:

```
/* in bs.h: */
typedef struct bs_string { /* ... */ } bs_string;    /* Bjarne's string */
typedef int bs_bool;      /* Bjarne's Boolean type */

/* in pete.h: */
typedef char* pete_string; /* Pete's string */
typedef char pete_bool;    /* Pete's Boolean type */
```

这种方法实在是太流行了, 因此使用一两个字母的前缀不是好的选择, 很容易与其他人代码中的名字产生冲突。

27.4 动态内存分配

C 并未提供 new 和 delete 操作符来进行对象分配与释放。为了实现动态内存分配, 你可以使用一些函数来直接分配和释放内存空间。最重要的几个函数定义在“一般工具”标准头文件 <stdlib.h> 中:

```
void* malloc(size_t sz);      /* allocate sz bytes */
void free(void* p);           /* deallocate the memory pointed to by p */
void* calloc(size_t n, size_t sz); /* allocate n*sz bytes initialized to 0 */
void* realloc(void* p, size_t sz); /* reallocate the memory pointed to by p
                                   to a space of size sz */
```

typedef size_t 也是在 <stdlib.h> 中定义的, 它是用 typedef 定义的无符号整型。

为什么 malloc() 返回一个 void*? 原因在于它不知道你要在分配到的内存空间中存入什么样

的对象。对象的初始化应该是你的责任，例如：

```
struct Pair {
    const char* p;
    int val;
};

struct Pair p2 = {"apple", 78};
struct Pair* pp = (struct Pair*) malloc(sizeof(Pair));    /* allocate */
pp->p = "pear";      /* initialize */
pp->val = 42;
```

注意，无论是在 C 中，还是在 C++ 中，都不可以这样写：

```
*pp = {"pear", 42};    /* error: not C or C++98 */
```

但在 C++ 中，你可以为 Pair 定义一个构造函数，然后这样写：

```
Pair* pp = new Pair("pear", 42);
```

在 C 中(C++ 中不可以，参见 27.3.4 节)，可以省略 malloc() 之前的类型转换，但我们不推荐这么做：

```
int* p = malloc(sizeof(int)*n);    /* avoid this */
```

省去类型转换的做法很流行，因为可以省去一些打字工作量，而且这样做还能检查到一种罕见的错误：在使用 malloc() 之前忘记了包含 <stdlib.h>。但是，这种方法也会掩盖内存大小的计算错误：

```
p = malloc(sizeof(char)*m);    /* probably a bug — not room for m ints */
```

不要在 C++ 程序中使用 malloc()/free()，因为 new/delete 不需要类型转换，可以进行初始化(构造函数)和清理工作(析构函数)，还能报告内存分配错误(抛出异常)，而且和 malloc()/free() 一样快。例如：

```
int* p = new int[200];
// ...
free(p);    // error

X* q = (X*)malloc(n*sizeof(X));
// ...
delete q;    // error
```

这段代码也许会正确运行，但它难以移植。而且，对于具有构造函数和析构函数的对象，如果混合使用 C 风格和 C++ 风格的动态内存分配代码，很容易导致灾难性后果。

函数 realloc() 通常用于扩展缓冲区：

```
int max = 1000;
int count = 0;
int c;
char* p = (char*)malloc(max);
while ((c=getchar())!=EOF) {    /* read: ignore chars on eof line */
    if (count==max-1) {        /* need to expand buffer */
        max += max;          /* double the buffer size */
        p = (char*)realloc(p,max);
        if (p==0) quit();
    }
    p[count++] = c;
}
```

C 的输入操作的相关内容，可参见 27.6.2 节和附录 B.10.2。

函数 realloc() 可以将数据从旧的内存空间移动到新的内存空间，但这种数据移动也有可能无法完成。绝对不要将 realloc() 用于 new 分配的内存空间。

在 C++ 中，(大致)等价的代码如下所示(使用了标准库)：

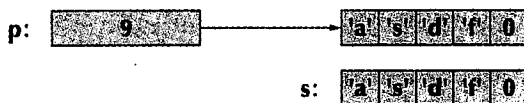
```
vector<char> buf;
char c;
while (cin.get(c)) buf.push_back(c);
```

请参考《Learning Standard C++ as a New Language》一文(参见 27.1 节中给出的参考文献列表),其中对输入和内容分配策略进行了全面的讨论。

27.5 C 风格字符串

在 C 中,字符串(在 C++ 的文献中,通常称为 C 字符串或 C 风格字符串)就是一个以 0(数值)结尾的字符数组。例如:

```
char* p = "asdf";
char s[] = "asdf";
```



在 C 中,没有成员函数机制,不能重载函数,也不能为 struct 定义运算符(如 ==)。因此我们需要一组函数(非成员函数)来处理字符串。C 和 C++ 的标准库提供了如下函数(在 <string.h> 中):

```
size_t strlen(const char* s);           /* count the characters */
char* strcat(char* s1, const char* s2); /* copy s2 onto the end of s1 */
int strcmp(const char* s1, const char* s2); /* compare lexicographically */
char* strcpy(char* s1, const char* s2); /* copy s2 into s1 */

char* strchr(const char* s, int c);      /* find c in s */
char* strstr(const char* s1, const char* s2); /* find s2 in s1 */

char* strncpy(char*, const char*, size_t n); /* strcpy, max n chars */
char* strncat(char*, const char, size_t n); /* strcat with max n chars */
int strncmp(const char*, const char*, size_t n); /* strcmp with max n chars */
```

这里并没有列出所有字符串函数,但最有用和最常用的函数都已经列出了。我们简单说明一下如何使用这些函数。

C 支持字符串比较。但相等运算符(==)比较的是两个字符串的指针值,标准库函数 strcmp() 才是比较字符串内容的:

```
const char* s1 = "asdf";
const char* s2 = "asdf";

if (s1==s2) { /* do s1 and s2 point to the same array? */
    /* (typically not what you want) */
}

if (strcmp(s1,s2)==0) { /* do s1 and s2 hold the same characters? */
}
}
```

函数 strcmp() 对字符串进行三路比较。如上面程序所示,对于给定的字符串参数 s1 和 s2, strcmp(s1,s2) 返回 0 表示两个字符串完全相等。如果在字典序中, s1 位于 s2 之前, strcmp 会返回一个负数,如果 s1 位于 s2 之后,返回一个正数。例如:

```
strcmp("dog","dog")==0
strcmp("ape","dodo")<0 /* "ape" comes before "dodo" in a dictionary */
strcmp("pig","cow")>0 /* "pig" comes after "cow" in a dictionary */
```

字符串指针的比较 s1 == s2 也不一定得到 0(false)。因为某些实现可能将所有相同内容的字符串只

保存一份, 这样 s1 和 s2 会指向相同内存空间, 于是比较的结果是 1(true)。因此, 使用 strcmp() 才是正确的 C 风格字符串比较方法。

函数 strlen() 用来获得 C 风格字符串的长度:

```
int lgt = strlen(s1);
```

注意, 长度不包括结尾的 0。在本例中, strlen(s1) == 4, 但 s1 实际占用了 5 个字节来保存“asdf”。这个微小的差别是很多差一位错误的根源。

我们还可以复制 C 风格字符串(结尾的 0 也会被复制):

```
strcpy(s1,s2); /* copy characters from s2 into s1 */
```

你(调用者)应该保证目标字符串(数组)有足够的空间容纳源字符串中的字符。

函数 strncpy()、strncat() 和 strncmp() 是 strcpy()、strcat() 和 strcmp() 限定处理长度的版本, 第三个参数 n 指出了最多处理多少个字符。注意, 如果源字符串中的字符不足 n 个, strncpy() 不会复制结尾的 0, 因此得到的结果是一个非法的 C 风格字符串。

函数 strchr() 和 strstr() 在第一个参数(一个字符串)中搜索第二个参数(字符或字符串), 并返回匹配的位置。与 find() 类似, 它们从左至右搜索。

令人惊奇的是这些简单的函数能完成很多功能, 同样令人惊奇的是使用这些函数非常容易出现小的错误。考虑这么一个简单问题: 将用户名和地址字符串连接起来, 之间加入一个 @。在 C++ 中, 可以使用 std::string:

```
string s = id + '@' + addr;
```

使用 C 风格字符串, 我们可以这样写:

```
char* cat(const char* id, const char* addr)
{
    int sz = strlen(id)+strlen(addr)+2;
    char* res = (char*) malloc(sz);
    strcpy(res,id);
    res[strlen(id)+1] = '@';
    strcpy(res+strlen(id)+2,addr);
    res[sz-1]=0;
    return res;
}
```

这段代码是正确的吗? 会有人释放 cat() 返回的字符串吗?

试一试 测试 cat()。为什么计算新字符串的长度时要加 2? 我们在 cat() 留下了一个初学者常犯的错误, 找到并修正它。我们“忘记”写注释了。请添加注释, 假定读者了解标准 C 字符串函数。

27.5.1 C 风格字符串和 const

考虑下面代码:

```
char* p = "asdf";
p[2] = 'x';
```

这段代码在 C 中是合法的, 但在 C++ 中不合法。在 C++ 中, 一个字符串文字常量被视为常量, 是不可更改的, 因此 p[2] = 'x' (将 p 指向的内容改为“asxf”)是非法的。不幸的是, 很少有编译器能捕获这个错误, 导致出现问题。如果你足够幸运的话, 程序运行时会产生一个错误, 但你不能期望总是这么幸运, 有可能产生更为严重的后果。你应该这样编写代码:

```
const char* p = "asdf"; // now you can't write to "asdf" through p
```

我们建议在 C 和 C++ 中都这样编写程序。

C 的 `strchr()` 函数有一个相似但更难以发现的问题。考虑如下代码：

```
char* strchr(const char* s, int c); /* find c in constant s (not C++) */

const char aa[] = "asdf";          /* aa is an array of constants */
char* q = strchr(aa, 'd');          /* finds 'd' */
*q = 'x';                          /* change 'd' in aa to 'x' */
```

这段代码在 C 和 C++ 中都是非法的，但 C 编译器不能捕获这个错误。这种错误优势被称为递变 (transmutation)：它把 `const` 类型变为非 `const` 类型，违反了对代码的合理假设。

在 C++ 中，可以用标准库声明的另一个 `strchr()` 函数：

```
char const* strchr(const char* s, int c); // find c in constant s
char* strchr(char* s, int c);            // find c in s
```

函数 `strstr()` 也存在同样问题。

27.5.2 字节操作

在遥远的黑暗时代(1980 年代早期)，当时 `void*` 尚未发明，C(和 C++) 程序员只能使用字符串操作来处理字节。现在的标准库中已经有了基本的内存处理函数，它们接受 `void*` 型参数，返回 `void*` 型值，以此来警告用户——它们直接处理内存，因此本质上讲处理对象应该是无类型的内存数据。

```
/* copy n bytes from s2 to s1 (like strcpy): */
void* memcpy(void* s1, const void* s2, size_t n);

/* copy n bytes from s2 to s1 ( [s1:s1+n) may overlap with [s2:s2+n) ): */
void* memmove(void* s1, const void* s2, size_t n);

/* compare n bytes from s2 to s1 (like strcmp): */
int memcmp(const void* s1, const void* s2, size_t n);

/* find c (converted to an unsigned char) in the first n bytes of s: */
void* memchr(const void* s, int c, size_t n);

/* copy c (converted to an unsigned char)
   into each of the first n bytes that s points to: */
void* memset(void* s, int c, size_t n);
```

不要在 C++ 中使用这些函数。特别是 `memset()`，它会干扰构造函数的正常工作。

27.5.3 实例：strcpy()

`strcpy()` 的定义应该是为人们所熟知的了，但它作为 C(和 C++) 简洁性范例的一面，就不那么为人所知了：

```
char* strcpy(char* p, const char* q)
{
    while (*p++ = *q++);
    return p;
}
```

为什么这段代码的确将 C 风格字符串 `q` 的内容复制到 `p` 中，留给大家思考。

试一试 这个 `strcpy()` 的实现正确吗？解释为什么。

如果你不能解释为什么，我们认为你还不是一个 C 程序员(不过你可能是合格的其他语言的程序员)。每种语言都有自己的风格特色，这就是 C 的特色。

27.5.4 一个风格问题

对一个常常引起激烈争论的，很大程度上与程序设计本身无关的风格问题，我们已经默默地

支持很长时间了。我们可以定义指针类型如下：

```
char* p;    // p is a pointer to a char
```

而不是这样定义：

```
char *p;    /* p is something that you can dereference to get a char */
```

空格放在哪里对于编译器来说毫无意义，但是程序员却很在意。我们的风格（在 C++ 中很常见）强调变量的类型，而第二种风格（在 C 中很常见）强调对指针变量的使用。注意，我们并不推荐在一条语句中声明很多变量：

```
char c, *p, a[177], *f();    /* legal, but confusing */
```

这种声明语句在老式程序中并不罕见。我们建议用多条语句来声明这些变量，并利用每行剩余的空间添加注释和初始化代码：

```
char c = 'a';    /* termination character for input using f() */
char* p = 0;    /* last char read by f() */
char a[177];    /* input buffer */
char* f();    /* read into buffer a; return pointer to first char read */
```

而且，应该为变量取更有意义的名字。

27.6 输入/输出：stdio

C 中没有 `iostream`，因此我们使用 `<stdio.h>` 中定义的 C 标准 I/O，这组特性通常称为标准 I/O (stdio)。stdio 中与 `cin` 和 `cout` 等价的是 `stdin` 和 `stdout`。在一个程序中，可以（对相同的 I/O 流）混合使用 `stdio` 和 `iostream`，但我们不推荐这么用。如果你觉得需要混合使用，请查阅专家级的参考书籍中有关 `stdio` 和 `iostream`（特别是 `ios_base::sync_with_stdio()`）的详细介绍。参见附录 B.10。

27.6.1 输出

最常用也最有用的 `stdio` 函数是 `printf()`，它的最基本的用途是打印（C 风格）字符串：

```
#include<stdio.h>
```

```
void f(const char* p)
{
    printf("Hello, World!\n");
    printf(p);
}
```

这个例子不是那么有趣。有趣的一点是 `printf()` 能够接受任意数量的参数，第一个参数（一个字符串）控制其后的参数如何打印。C 中 `printf()` 的定义如下所示：

```
int printf(const char* format, ...);
```

“...”的含义是“可以有更多参数”。我们可以这样调用 `printf()`：

```
void f1(double d, char* s, int i, char ch)
{
    printf("double %g string %s int %d char %c\n", d, s, i, ch);
}
```

这里，`%g` 表示“用一般格式打印一个浮点值”，`%s` 表示“打印一个 C 风格字符串”，`%d` 表示“以十进制格式打印一个整数”，`%c` 表示“打印一个字符”。每个格式限定符都打印下一个未处理的参数，因此 `%g` 打印 `d`，`%s` 打印 `s`，`%d` 打印 `i`，`%c` 打印 `ch`。附录 B.10.2 中给出了 `printf()` 的完整格式限定符列表。

不幸的是，`printf()` 不是类型安全的，例如：

```
char a[] = { 'a', 'b' };    /* no terminating 0 */

void f2(char* s, int i)
{
    printf("goof %s\n", i);    /* uncaught error */
    printf("goof %d: %s\n", i); /* uncaught error */
    printf("goof %s\n", a);    /* uncaught error */
}
```

最后一个 printf() 的效果很有趣：它会打印 a[1] 之后内存中的每个字节(字符)，直至遇到 0 为止。打印出的字符数目可能非常非常多。

虽然 stdio 在 C 和 C++ 中都能正常工作，但由于缺少类型检查，我们更倾向于使用 iostream。倾向于 iostream 的另一个原因是 stdio 函数没有扩展性：你无法扩展 printf() 来输出自定义类型，而使用 iostream 是可以做到这点的。例如，你没办法定义新的格式限定符 %Y 来输出自定义类型 struct Y。

printf() 还有一个很有用的版本，可以向文件中打印数据：

```
int fprintf(FILE* stream, const char* format, ...);
```

例如：

```
fprintf(stdout, "Hello, World!\n"); // exactly like printf("Hello, World!\n");
FILE* ff = fopen("My_file", "w"); // open My_file for writing
fprintf(ff, "Hello, World!\n");    // write "Hello, World!\n" to My_file
```

第一个参数是一个文件句柄(文件描述符)，文件句柄将在 27.6.3 节中介绍。

27.6.2 输入

最常用的 stdio 输入函数包括：

```
int scanf(const char* format, ...); /* read from stdin using a format */
int getchar(void);                 /* get a char from stdin */
intgetc(FILE* stream);             /* get a char from stream */
char* gets(char* s);               /* get characters from stdin */
```

最简单的读取字符串的方式是使用 gets()，例如：

```
char a[12];
gets(a); /* read into char array pointed to by a until a '\n' is input */
```

但是，不要这样编写程序！gets() 是有害的！曾经有大约 1/4 的成功黑客攻击是由于 gets() 和它的近亲 scanf("%s") 的漏洞造成的。到现在为止，这仍然是一个主要的安全问题。以上面简单的程序为例，你如何知道在换行之之前用户最多输入 11 个字符呢？你是无法知道的。因此，gets() 几乎肯定会导致内存破坏(缓冲区之后的内存空间)，而内存破坏目前仍是黑客的主要工具之一。不要认为你可以猜测一个最大缓冲区规模，能“对所有用户都足够大”。也许在输入流另一端的那个“人”只是一个程序，它会打破你的合理假设。

函数 scanf() 使用类似 printf() 的格式限定串来指定输入格式。其使用与 printf() 一样方便：

```
void f()
{
    int i;
    char c;
    double d;
    char* s = (char*)malloc(100);
    /* read into variables passed as pointers */
    scanf("%i %c %g %s", &i, &c, &d, s);
    /* %s skips initial whitespace and is terminated by whitespace */
}
```

与 printf() 类似，scanf() 也不是类型安全的。格式字符串和参数(指针)必须严格匹配，否则在运

行时就会产生奇怪的结果。而且, %s 将字符串读入 s 的过程中还会发生溢出。因此, 永远不要使用 gets() 或 scanf("%s")!

那么如何才能安全地读取字符呢? 我们可以在格式限定符 %s 中指定要读取的字符的数目, 例如:

```
char buf[20];
scanf("%19s", buf);
```

我们需要为结尾的 0 留出存储空间, 因此能读入 buf 的最大字符数为 19。但是, 这又引起一个新的问题: 如果用户输入的字符数超过了 19 个, 应该怎么办呢? scanf() 的处理方式是将“多余”的字符留在输入流中, 随后的输入操作可能会“发现”这些字符。

由于 scanf() 存在这些问题, 一般来说更为谨慎也更为容易的方法是使用 getchar()。使用 getchar() 读取字符的一般方法如下:

```
while((x=getchar())!=EOF) {
    /* ... */
}
```

EOF 是一个 stdio 宏, 它表示“文件尾”的含义, 参见 27.4 节。

C++ 标准库中的流与 scanf("%s") 和 get() 功能类似, 但不存在上述问题:

```
string s;
cin >> s;      // read a word
getline(cin,s); // read a line
```

27.6.3 文件

在 C(或 C++) 中, 可以使用 fopen() 打开文件, 使用 fclose() 关闭文件。这些函数与文件描述符结构 FILE 及 EOF 宏(文件尾)都定义在 <stdio.h> 中:

```
FILE *fopen(const char* filename, const char* mode);
int fclose(FILE *stream);
```

你可以这样使用文件:

```
void f(const char* fn, const char* fn2)
{
    FILE* fi = fopen(fn, "r");    /* open fn for reading */
    FILE* fo = fopen(fn2, "w");   /* open fn for writing */

    if (fi == 0) error("failed to open input file");
    if (fo == 0) error("failed to open output file");

    /* read from file using stdio input functions, e.g., getc() */
    /* write to file using stdio output functions, e.g., fprintf() */

    fclose(fo);
    fclose(fi);
}
```

考虑这样一个问题: C 中没有异常机制, 那么在发生错误的情况下, 我们如何保证文件确实被关闭了?

27.7 常量和宏

在 C 中, const 绝不是编译时常量:

```
const int max = 30;
const int x; /* const not initialized: OK in C (error in C++) */

void f(int v)
{
    int a1[max]; /* error: array bound not a constant (OK in C++) */
                /* (max is not allowed in a constant expression!) */
    int a2[x];   /* error: array bound not a constant */
}
```

```

switch (v) {
case 1:
    /* ... */
    break;
case max: /* error: case label not a constant (OK in C++) */
    /* ... */
    break;
}

```

在 C 中这样规定(与 C++ 不同),是因为考虑到技术上的原因: `const` 隐含地可被所有源文件访问,例如:

```

/* file x.c: */
const int x; /* initialize elsewhere */

/* file xx.c: */
const int x = 7; /* here is the real definition */

```

在 C++ 中,这是两个不同的对象,名字都是 `x`,作用域在自己的文件中。C 程序员不是用 `const` 来表示符号常量,他们更愿意使用宏。例如:

```

#define MAX 30
void f(int v)
{
    int a1[MAX]; /* OK */

    switch (v) {
case 1:
    /* ... */
    break;
case MAX: /* OK */
    /* ... */
    break;
}
}

```

程序中凡是使用宏 `MAX` 的地方,都被替换为文本 30(宏的值)。也就是说, `a1` 的元素数目为 30,第二个 `case` 语句的值也是 30。我们为宏取名时使用了全部大写的字符串 `MAX`,这是 C 语言的惯例。这种命名习惯有助于减少宏引起的错误。

27.8 宏

使用宏的时候一定要小心:在 C 中没有真正有效的方法来避免使用宏,但宏带有严重的副作用,因为宏不遵守通常的 C(或 C++)作用域和类型规则——它只是一种文本替换而已。参见附录 A.17.2。

除了尽量不用宏(使用 C++ 中的替代方法)之外,我们还有什么办法来避免宏引起的问题吗?

- 所有宏名都全部大写。
- 不是宏的结构不要使用全部大写的名字。
- 不要为宏取短的或“有趣”的名字,如 `max` 或 `min`。
- 期望其他人也遵守上述简单而常见的规范。

宏的主要用途包括:

- 定义“常量”。
- 定义类似函数的结构。

- “改进”语法。
- 控制条件编译。

另外还有其他很多不太常见的用途。

我们认为宏被过度使用了，但在 C 程序中，还没有一种合理而完整的替代方法。甚至在 C++ 程序中也很难避免使用宏（特别是当你编写的程序需要移植到很老的编译器上或者有特殊限制的平台时）。

对于那些认为下面介绍的技术是“低级手段”，不应该在此提及的人，我们要说声抱歉了。因为我们认为这种编程技术是现实世界中真实存在的，而且我们所选择的这些（很温和的）例子展示了宏的正确使用和不正确使用，可以帮助初学者避免长时间陷入困境。对宏的无知会带来不幸。

27.8.1 类函数宏

下面是一个非常典型的类函数宏：

```
#define MAX(x, y) ((x)>=(y)?(x):(y))
```

我们为宏取名为全大写字母的 MAX，以便与（各种程序中）常用的函数名 max 区别开来。显然，它与函数还是有很大区别的：没有参数类型、没有语句块、没有返回语句等。另外，宏定义中的那些括号是起什么作用的？考虑如下代码：

```
int aa = MAX(1,2);
double dd = MAX(aa++,2);
char cc = MAX(dd,aa)+2;
```

宏替换后，程序扩展为：

```
int aa = ((1)>=( 2)?(1):(2));
double dd = ((aa++)>=(2)?( aa++):(2));
char cc = ((dd)>=(aa)?(dd):(aa))+2;
```

如果在宏定义中没有使用“那些括号”，最后一条语句会扩展为：

```
char cc = dd>=aa?dd:aa+2;
```

也就是说，cc 的值将和你根据其定义推断出的值不同。这个例子说明，在宏的定义中，使用任何参数时都应将其置于括号之中（当做表达式）。

另一方面，对于第二条语句，使用再多括号也解决不了问题。宏参数 x 被替换为 aa ++，由于 x 在 MAX 使用了两次，因此 x 进行了两次增 1 运算。记住，不要向宏传递可能引起副作用的参数。

某些天才可能碰巧定义了这样有问题的宏，并将其放入了被广泛使用的头文件中。更不幸的是，他还将宏命名为 max 而不是 MAX，这样，当 C++ 标准头文件中定义下面函数时

```
template<class T> inline T max(T a,T b) { return a<b?b:a; }
```

max(T a, T b) 就会被扩展，在编译器看来，语句变为：

```
template<class T> inline T ((T a)>=( T b)?( T a):( T b)) { return a<b?b:a; }
```

编译器给出的错误信息会非常“有趣”，对程序员修正错误毫无帮助。如果遇到这种紧急情况，你可以“取消定义”（undefine）宏：

```
#undef max
```

幸运的是，这个宏并不是那么重要。但是，在那些广泛应用的头文件中有成千上万个宏，不可能取消每个宏都不引起混乱。

并不是所有宏参数都被用作表达式，例如：

```
#define ALLOC(T,n) ((T*)malloc(sizeof(T)*n))
```

这是来自实际程序中的例子，内存分配时 sizeof 中使用的类型与所需类型可能不匹配，这个宏对避免此类错误很有用：

```
double* p = malloc(sizeof(int)*10); /* likely error */
```

不幸的是,如果希望宏还能捕获内存耗尽错误,就不那么好办了。假如已经定义了 `error_var` 和 `error()`,可以这样定义宏:

```
#define ALLOC(T,n) (error_var = (T*)malloc(sizeof(T)*n),\
                    (error_var==0)\
                    ?(error("memory allocation failure"),0)\
                    :error_var)
```

行结尾处的“\”并非输入错误,将一个较长的宏分成几行,就必须在非结尾行的最后加上“\”。如果你使用 C++ 编写程序,我们建议还是使用 `new`。

27.8.2 语法宏

你可以定义这样一类宏,它们能使源程序形式上更符合你的偏好,例如:

```
#define forever for(;;)
#define CASE break; case
#define begin {
#define end }
```

我们强烈建议不要使用这种宏。很多人已经尝试过这种方法了。他们(以及他们编写出的代码的维护人员)发现:

- 对于“好的语法”,很多人的理解是不一样的。
- “改进的语法”是不标准的、奇怪的,会令他人困惑。
- 有过“改进语法”导致难以发现的编译错误的先例。
- 你所看到的并非编译器所看到的,编译器是根据它所知道的(以及在源程序中所看到的)词汇报告错误,而不是根据你所知及你所见。

因此,不要使用语法宏“改进”代码外观。你和你的好朋友可能觉得效果很棒,但经验表明,你只是大社区中的一分子而已,因而其他人将不得不重写你的代码(假如你的代码还“活着”的话)。

27.8.3 条件编译

假设某个头文件有两个版本,比如说一个是 Linux 版,另一个是 Windows 版。在程序中你如何选择使用哪个版本呢?常用方法如下:

```
#ifdef WINDOWS
#include "my_windows_header.h"
#else
#include "my_linux_header.h"
#endif
```

现在,如果有人在编译之前定义了宏 `WINDOWS`,则效果为:

```
#include "my_windows_header.h"
```

否则,效果为:

```
#include "my_linux_header.h"
```

`#ifdef WINDOWS` 并不关心 `WINDOWS` 被定义为什么,它只关心 `WINDOWS` 是否被定义。

很多大型系统(包括所有操作系统)都会定义类似 `WINDOWS` 这样的宏,以供你检测。我们可以这样来检测程序是在被 C++ 编译器编译还是在被 C 编译器编译:

```
#ifdef __cplusplus
// in C++
#else
/* in C */
#endif
```

还有一种类似的结构,通常被人们称为包含保护(include guard),常常用来防止头文件被包含多次:

```

/* my_windows_header.h: */
#ifndef MY_WINDOWS_HEADER
#define MY_WINDOWS_HEADER
    /* here is the header information */
#endif

```

`#ifndef` 检测宏是否未被定义，即它与 `#ifdef` 是相对的。逻辑上，用于源文件控制的宏与其他修改源码（宏替换）的宏有很大不同。它们只是使用了相同的下层语言机制而已。

27.9 实例：侵入式容器

C++ 标准库容器（如 `vector` 和 `map`）是非侵入式容器（non-intrusive container）：即它们不要求容器内的数据以单个元素的形式被访问，而是对容器整体进行操作。这也是它们为什么有那么好的通用性——适用于所有内置类型和用户自定义类型，只要类型支持复制操作即可。另外一类容器被称为侵入式容器（intrusive container），在 C 和 C++ 中都很常用。下面我们将通过一个非侵入式的容器来说明 C 风格 `struct`、指针和动态内存分配的使用。

我们可以定义一个支持如下 9 个操作的双向链表：

```

void init(struct List* lst);      /* initialize lst to empty */
struct List* create();           /* make a new empty list on free store */
void clear(struct List* lst);    /* free all elements of lst */
void destroy(struct List* lst);  /* free all elements of lst, then free lst */

void push_back(struct List* lst, struct Link* p); /* add p at end of lst */
void push_front(struct List*, struct Link* p);   /* add p at front of lst */

/* insert q before p in lst: */
void insert(struct List* lst, struct Link* p, struct Link* q);
struct Link* erase(struct List* lst, struct Link* p); /* remove p from lst */

/* return link n "hops" before or after p: */
struct Link* advance(struct Link* p, int n);

```

基本设计思路是让用户只需提供 `List *` 和 `Link *` 指针就能完成这些操作。这意味着可以大幅度修改这些操作的实现，而无需修改用户程序。显然，我们在命名上受到了 STL 的影响。`List` 和 `Link` 显然可以简单定义如下：

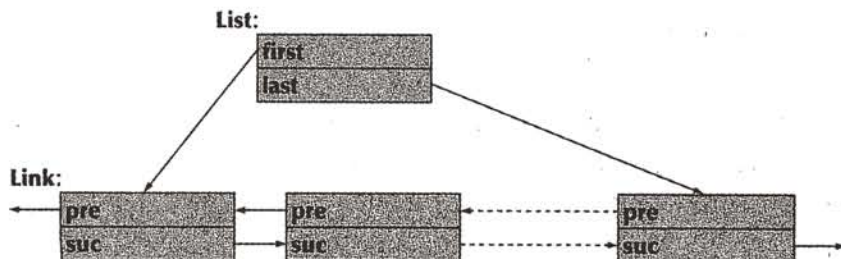
```

struct List {
    struct Link* first;
    struct Link* last;
};

struct Link { /* link for doubly-linked list */
    struct Link* pre;
    struct Link* suc;
};

```

下面是 `List` 的图示：



我们目的不是介绍高明的描述技术或者高明的算法，因此本节并未展示这些。但是，请注意程序中并未提及 Link 所保存的数据 (List 中的元素)。回过头看一下程序，我们发现 Link 和 List 非常像抽象类。Link 保存的数据会随后提供。Link* 和 List* 有时称为不透明类型处理工具，即我们可以在不了解 Link 和 List 的内部结构的情况下，使用 Link* 和 List* 来处理 List 中的元素。

为了实现 List 函数，我们首先要#include 一些标准库头文件：

```
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>
```

C 不支持名字空间，因此我们不必担心 using 声明或者 using 指令的问题。另一方面，我们应该担心的可能是那些非常常见的简单名字 (Link、insert、init 等)，因此这组函数不应在这个玩具程序之外使用。

初始化代码很简单，但注意 assert() 的使用：

```
void init(struct List* lst) /* initialize *p to the empty list */
{
    assert(lst);
    lst->first = lst->last = 0;
}
```

我们决定在运行时不处理非法链表指针错误。通过使用 assert()，我们只是对空链表指针给出一个(运行时)系统错误。“系统”错误会给出失败的 assert() 所在的文件名和行号；assert() 是在 <assert.h> 中定义的宏，其检测只在调试状态下才执行。由于 C 语言不支持异常，处理非法指针是很困难的。

函数 create() 简单地在动态内存空间中创建一个 List。它在某种程度上可以看做构造函数 (init() 进行初始化) 和 new (malloc() 完成内存分配)：

```
struct List* create() /* make a new empty list */
{
    struct List* lst = (struct List*)malloc(sizeof(struct List));
    init(lst);
    return lst;
}
```

函数 clear() 假定所有 Link 的内存空间都是动态分配的，因此用 free() 来释放：

```
void clear(struct List* lst) /* free all elements of lst */
{
    assert(lst);
    {
        struct Link* curr = lst->first;
        while(curr) {
            struct Link* next = curr->suc;
            free(curr);
            curr = next;
        }
        lst->first = lst->last = 0;
    }
}
```

注意我们使用 Link 成员 suc 的方法。如果一个对象已经被释放了，我们是无法安全访问其成员的。因此，在释放一个 Link 时，我们引入变量 next，保存所处的列表位置。

如果我们并不是在动态内存空间中分配全部 Link，那么最好不要调用 clear() 来释放链表内存空间，否则会造成很大的混乱。

destroy() 本质上与 create() 是相对的, 也就是说, 它是析构函数和 delete 的结合:

```
void destroy(struct List* lst) /* free all elements of lst; then free lst */
{
    assert(lst);
    clear(lst);
    free(lst);
}
```

注意, 我们没有准备为链表元素调用清理函数(析构函数)。也就是说, 目前的设计并非是 C++ 技术和普遍方法的准确模拟, 实际上, 我们不能也不必这样做。

函数 push_back() 的设计思路非常直接——向链表中添加一个 Link, 作为新的表尾:

```
void push_back(struct List* lst, struct Link* p) /* add p at end of lst */
{
    assert(lst);
    {
        struct Link* last = lst->last;
        if (last) {
            last->suc = p; /* add p after last */
            p->pre = last;
        }
        else {
            lst->first = p; /* p is the first element */
            p->pre = 0;
        }
        lst->last = p; /* p is the new last element */
        p->suc = 0;
    }
}
```

但是, 如果我们不在草稿纸上画一些方块(链表节点)和箭头(链表指针), 来分析链表的操作方式, 是很难直接写出正确的代码的。注意, 在上述代码中, 我们“忘记”考虑了参数 p 为空的情况。将 0 而不是一个合法指针传递给 Link, 这段代码就会崩溃。这段代码谈不上糟糕, 但它不是一个工业级别的代码。其目的是说明常用的、有用的技术, 在本例中, 它的另一目的是展示一种常见的弱点/bug。

函数 erase() 可以这样编写:

```
struct Link* erase(struct List* lst, struct Link* p)
/*
    remove p from lst;
    return a pointer to the link after p
*/
{
    assert(lst);
    if (p==0) return 0; /* OK to erase(0) */

    if (p == lst->first) {
        if (p->suc) {
            lst->first = p->suc; /* the successor becomes first */
            p->suc->pre = 0;
            return p->suc;
        }
        else {
            lst->first = lst->last = 0; /* the list becomes empty */
            return 0;
        }
    }
}
```

```

else if (p == lst->last) {
    if (p->pre) {
        lst->last = p->pre; /* the predecessor becomes last */
        p->pre->suc = 0;
    }
    else {
        lst->first = lst->last = 0; /* the list becomes empty */
        return 0;
    }
}
else {
    p->suc->pre = p->pre;
    p->pre->suc = p->suc;
    return p->suc;
}
}
}

```

我们将剩余函数的编写作为练习，在我们的非常简单的测试中也用不到这些函数。但是，现在我们必须面对目前设计中最费解的一个问题：链表元素中的数据在哪里？例如，我们想实现一个保存名字(C 风格字符串)的链表，应该怎么做？考虑下面代码：

```

struct Name {
    struct Link link; /* the Link required by List operations */
    char* p; /* the name string */
};

```

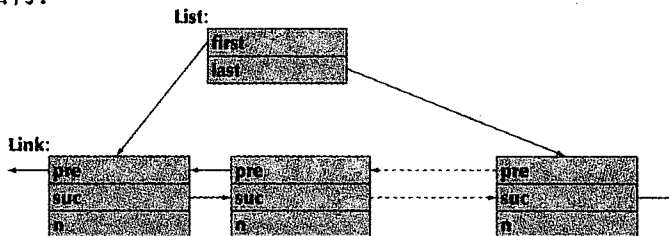
到目前为止，一切尚好，只是我们还没弄清如何使用 Name 的 Link 成员。不过由于我们知道 List 希望其中的 Link 在动态空间中分配内存，因此我们可以编写函数，在动态空间中创建 Name：

```

struct Name* make_name(char* n)
{
    struct Name* p = (struct Name*)malloc(sizeof(struct Name));
    p->p = n;
    return p;
}

```

下图描述了链表的结构：



下面程序展示了其使用方式：

```

int main()
{
    int count = 0;

    struct List names; /* make a list */
    struct List* curr;
    init(&names);

    /* make a few Names and add them to the list: */
    push_back(&names, (struct Link*)make_name("Norah"));
    push_back(&names, (struct Link*)make_name("Annemarie"));
    push_back(&names, (struct Link*)make_name("Kris"));

    /* remove the second name (with index 1): */
    erase(&names, advance(names.first, 1));
}

```

```

curr = names.first; /* write out all names */
for (; curr!=0; curr=curr->suc) {
    count++;
    printf("element %d: %s\n", count, ((struct Name*)curr)->p);
}
}

```

可以看出，我们使用了“欺骗手段”。我们将 `Name*` 转换为 `Link*`。通过这种方式，用户程序能够获取“库类型”`Link`。然而，“库”却不会（也不必）知道“用户程序类型”`Name`。这种方法是允许的吗？是的，这是允许的：在 C（和 C++）中，你可以将一个 `struct` 指针当做其第一个成员的指针来处理，反之亦然。

显然，本例也是合法的 C++ 程序。

试一试 C++ 程序员常对 C 程序员说的一句话是：“你所能做的每件事，我都能做得更好！”请用 C++ 语言重写侵入式 List 程序，来展示如何用更短、更简单的代码实现相同的功能，又不会牺牲速度和内存空间。

简单练习

1. 用 C 语言编写“Hello World!”程序，编译、运行它。
2. 定义两个变量，分别保存“Hello”和“World!”，将两个字符串连接在一起，中间加入一个空格，并输出为“Hello World!”。
3. 定义一个 C 函数，它接受两个参数：一个 `char*` 类型，名为 `p`；另一个为 `int` 型，名为 `x`。函数输出两个参数的值，形式为：`p is "foo" and x is 7`。用一些实际参数来测试这个函数。

思考题

在下面的题目中，假定 C 表示 ISO 标准 C89。

1. C++ 是 C 的子集吗？
2. 谁发明了 C？
3. 说出一本获得极高声誉的 C 教科书？
4. C 和 C++ 是在哪个机构中发明出来的？
5. 为什么 C++ 与 C（几乎）兼容？
6. 为什么 C++ 只是与 C 几乎兼容？
7. 列出十几个 C 不支持的 C++ 特性？
8. 现在哪个组织“拥有”C 和 C++？
9. 列出 6 个 C 中无法使用的 C++ 标准库功能。
10. 哪些 C 标准库功能在 C++ 中使用？
11. 如何在 C 中实现函数参数类型检查？
12. 哪些 C++ 特性相关的函数在 C 中不存在？列出至少 3 个，并举实例。
13. 如何从 C++ 程序调用 C 函数？
14. 如何从 C 程序调用 C++ 函数？
15. 哪些类型的内存布局在 C 和 C++ 中是兼容的？请举例。
16. 什么是结构标签？
17. 列出 20 个 C 中不存在的 C++ 关键字。
18. “`int x;`”在 C++ 中是一个定义吗？在 C 中呢？
19. 什么是 C 风格类型转换？它为什么是危险的？
20. `void*` 是什么？它在 C 和 C++ 中有什么不同？
21. 枚举类型在 C 和 C++ 中有什么不同？
22. 在 C 中如何才能避免常用名字所引起的连接问题？

23. C 中最常用的动态内存空间相关函数是哪 3 个?
24. C 风格字符串的定义?
25. 对于 C 风格字符串, == 和 strcmp() 有何不同?
26. 如何复制 C 风格字符串?
27. 如何获得一个 C 风格字符串的长度?
28. 如何复制一个大的 int 数组?
29. printf() 的优点是什么? 它的问题/局限是什么?
30. 为什么永远不要使用 gets()? 替代方法是什么?
31. 在 C 中如何打开一个文件?
32. const 在 C 和 C++ 中有何区别?
33. 我们为什么不喜欢宏?
34. 宏的常见用途是什么?
35. 包含保护是什么?

术语

#define	Dennis Ritchie	非侵入式
#ifdef	FILE	不透明类型
#ifndef	fopen()	重载
贝尔实验室	格式字符串	printf()
Brain Kernighan	侵入式	strcpy()
C/C++	K&R	结构标签
兼容性	字典序	三路比较
条件编译	连接	void
C 风格类型转换	宏	void*
C 风格字符串	malloc()	

习题

对于本章的习题, 最好对所有程序都同时在 C 和 C++ 两种编译器下编译。如果只使用 C++ 编译器, 你可能无意中使用了 C 不支持的特性。如果只使用 C 编译器, 类型错误可能无法被检测到。

1. 实现 strlen(), strcmp() 和 strcpy()。
2. 将 27.9 节中的侵入式 List 程序补充完整, 并测试所有函数。
3. 尽可能地“美化”27.9 节中的侵入式 List 程序, 使之更易使用; 捕获/处理尽量多的错误。改变 struct 定义的细节, 使用宏等方法都是合理的。
4. 为 27.9 节中的侵入式 List 程序编写 C++ 版本, 并测试每个函数。
5. 比较习题 3 和习题 4 的结果。
6. 改变 27.9 节中 Link 和 List 的实现, 但不改变用户函数接口。在一个数组中为 Link 分配空间, 并将其成员 first、last、pre 和 suc 定义为 int 类型(数组下标)。
7. 与 C++ 标准库容器(非侵入式)相比, 侵入式容器的优点和缺点是什么? 列出优缺点。
8. 你的机器中的字典序是怎样的? 输出你的键盘上的每个字符及其整数值。然后, 按整数值的顺序输出所有字符。
9. 只使用 C 语言特性和 C 标准库, 从 stdin 读入一个单词序列, 然后按字典序将它们输出到 stdout。提示: C 中的排序函数称为 qsort(), 查找它是在哪里定义的, 使用它来完成题目。另一种方法是, 每读入一个单词, 就将其插入到已排序的列表中。C 标准库中未定义列表结构。
10. 列出从 C++ 语言或支持类的 C 语言中借鉴来的 C 语言特性。
11. 列出没有被 C++ 采纳的 C 特性。
12. 实现一个支持查找功能的表结构, 每个表项保存一个 C 风格(string, int 对), 支持 find(struct table*,

`const char*`)、`insert(struct table*, const char*, int)`以及 `remove(struct table*, const char*)` 操作。表可以用一个 `struct` 数组或者一对数组(`const char*` 和 `int*`)来保存,你可以选择其中一种方式。函数返回类型也由你选择。编写文档,将你的设计决策描述清楚。

13. 编写 C 程序,实现 `string s; cin >> s;` 相同的功能。也就是说,定义一个输入操作,读入任意长度的以空白符结尾的字符序列,存入以 0 结尾的 `char` 数组中。
14. 编写函数,接受一个 `int` 数组参数,找出其中的最小值和最大值,并计算均值和中值。使用一个 `struct` 保存结果,返回值就设定为这个 `struct`。
15. 在 C 中模拟出单重继承。令每个“基类”包含一个指向指针数组的指针(用一组独立函数模拟虚函数,每个函数的第一个参数为指向一个“基类”对象的指针),参考 27.2.3 节。“派生”机制实现方式为:将派生的第一个成员定义为“基类”类型。对每个类,恰当地对“虚函数”数组进行初始化。为了测试这种模拟方式,用它实现“Shape”,基类的派生类的 `draw()` 只是简单地打印类名。在完成本题的过程中,只允许使用标准 C 特性和 C 标准库功能。
16. 使用宏简化上一题中的符号。



附言

我们曾经提到,兼容性问题不那么令人兴奋。然而,已经有大量的(数十亿行)C 代码“在那里”了,如果你必须阅读或编写 C 代码,本章向你介绍了一些预备知识。从个人角度,我更倾向于使用 C++,本章的一些内容也给出了部分原因。请不要低估“侵入式 List”例程,“侵入式 List”和不透明类型在 C 和 C++ 中都是非常重要和强大的工具。

术 语 表

“通常，几个恰如其分的单词就抵得上上千幅图片。”

——匿名

术语表(glossary)是一本书中所使用的词汇的简单解释。本术语表相当简短，只列出了我们认为最基本的术语(特别是在学习程序设计的早期)。书中每章的“术语”小节也有相关的内容，会有所帮助。至于更完整的C++术语表，可以参考www.research.att.com/~bs/glossary.html，在互联网上你还可以找到大量(质量各异的)专门的术语表。请注意，一个术语可能具有多个相关的含义(因此我们偶尔会对一个术语列出多个意义)，而我们列出的大多数术语在其他领域中有(弱)相关的含义。例如，我们没有按现代绘画、法律实践以及哲学中的相关含义来定义抽象。

abstract class(抽象类) 不能用来直接创建对象的类；通常用于定义派生类的接口。通过定义一个纯虚函数或者一个受保护的构造函数，可以使类成为抽象类。

abstraction(抽象) 对某个事物的描述，有选择、有目的地忽略(隐藏)了细节(如实现细节)；选择性的忽略。

address(地址) 一个值，可以帮助我们在计算机内存中找到某个对象。

algorithm(算法) 用于解决某个问题的过程或方法，能生成结果的有限的计算步骤序列。

alias(别名) 引用对象的另一种方法；通常是一个名字、一个指针或一个引用。

application(应用) 一个程序或者一组程序，用户将其视为一个整体。

approximation(近似) 与完美或理想的结果(一个值或一个设计方案)接近的结果(值或设计)。

通常，近似是在理想情况间折衷的结果。

argument(实参) 传递给函数或模板的值，通过形参访问。

array(数组) 同构元素的序列，元素通常被编号，如[0: max)。

assertion(断言) 插入程序中的语句，声明(断言)在此处某些条件必须为真。

base class(基类) 在类层次中当做基使用的类。通常基类包含一个或多个虚函数。

bit(位) 计算机中的基本信息单元。一个位的值可以是0或1。

bug(错误) 程序中的错误。

byte(字节) 多数计算机中的基本寻址单元。一个字节通常包含8位。

class(类) 用户自定义类型，可以包含数据成员、成员函数和成员类型。

code(代码) 一个程序或一个程序片段。既可以指源代码，又可以指目标代码，因此容易引起歧义。

compiler(编译器) 将源代码转换为目标代码的计算机程序。

complexity(复杂性) 一个很难精确定义的概念/度量，描述构造一个问题的解决方案的困难程度，或者解决方案本身的困难程度。有时用来(简单地)表示对执行一个算法所需操作次数的估计。

computation(计算) 某段代码的执行过程，一般接受一些输入，产生一些输出。

concrete class(具体类) 可以用来创建对象的类。

- constant** (常量) (在给定作用域内)不可改变的值; 不变量。
- constructor** (构造函数) 初始化(“构造”)对象的操作。通常一个构造函数会建立起一个不变量, 并申请对象所需的资源(这些资源通常由析构函数释放)。
- container** (容器) 容纳元素(其他对象)的对象。
- correctness** (正确性) 当一个程序或一个程序片段满足规范时, 则它是正确的。不幸的是, 规范可能不完整或不一致, 或不满足用户的合理期望。因此, 为了生成可接受的代码, 有时不仅要遵循规范, 还要做得更多。
- cost** (代价) 生成一个程序或者执行一个程序的花费(如程序设计时间、运行时间或空间)。理想情况, 代价应是复杂性的函数。
- data** (数据) 计算中用到的值。
- debugging** (调试) 搜索、去除程序中错误的活动, 与测试相比, 通常缺乏系统性。
- declaration** (声明) 程序中对名字及其类型的说明。
- definition** (定义) 一个实体的声明, 对于使用这个实体的程序, 定义提供了它所需要的完整信息。简化定义: 分配内存的声明。
- derived class** (派生类) 从一个或多个基类派生出的类。
- design** (设计) 一个总体描述, 指出一个软件应该如何操作来满足其规范。
- destructor** (析构函数) 当对象销毁时(如在作用域尾)被隐式调用的操作。通常会释放资源。
- encapsulation** (封装) 保护那些想要作为私有内容的部分(如实现细节)不会被未授权者访问。
- error** (错误) 期望的程序行为(通常表述为需求或用户指南)与程序的实际表现不匹配。
- executable** (可执行) 可在计算机上运行(执行)的程序。
- feature creep** (功能蔓延) 向程序添加过多功能(而只是为了“以防万一”)的倾向。
- file** (文件) 计算机中保存持久信息的容器。
- floating-point number** (浮点数) 实数在计算机中的近似, 如 7.93 和 $10.78e-3$ 。
- function** (函数) 命名的代码单元, 可从程序中不同位置进行调用; 计算的逻辑单元。
- generic programming** (泛型程序设计) 一种程序设计风格, 关注算法的设计和高效实现。一个泛型算法可应用于所有满足其要求的实参类型。在 C++ 中, 泛型程序设计通常使用模板。
- header** (头文件) 包含声明的文件, 其中的声明用于在程序的不同部分之间共享接口。
- hiding** (隐藏) 阻止信息片段直接可见或直接被访问的动作。例如, 嵌入的(内层)作用域中的名字可以阻止外层作用域中的相同名字被直接使用。
- 理想 (ideal)** 我们追求的某个事物的完美结果。但通常难以获得, 我们不得不做出折衷, 接受一个近似结果。
- implementation** (实现) 1) 编写和测试代码的动作; 2) 实现某个程序的代码。
- infinite loop** (无限循环) 终止条件永远为假的循环。参见迭代(iteration)。
- infinite recursion** (无限递归) 直至机器内存耗尽才会结束的递归。在实际中, 这种递归不会是无限的, 会终止于某种硬件错误。
- information hiding** (信息隐藏) 将接口和实现分离的动作, 因此隐藏的实现细节对用户不可见, 提供了一个抽象。
- initialize** (初始化) 赋予对象一个初值。
- input** (输入) 计算所要使用的值(例如, 函数实参及用键盘敲入的字符)。

- integer(整数)** 如 42 和 -99。
- interface(接口)** 一个声明或一组声明, 指出一段代码(如一个函数或一个类)如何被调用。
- invariant(不变量)** 在程序某个位置(或某几个位置)必须始终为真的事物; 通常用来描述对象的状态(一组值)或循环进入重复语句前的状态。
- iteration(迭代)** 重复执行一段代码的动作; 参见递归(recursion)。
- iterator(迭代器)** 标识序列中元素的对象。
- library(库)** 一组类型、函数、类等, 实现了一组功能(抽象), 可能被很多程序所使用。
- lifetime(生命期)** 从对象初始化到其不可用(离开作用域、被释放或程序结束)之间的时间。
- linker(链接器)** 一个程序, 将目标代码文件和库组合在一起, 生成一个可执行程序。
- literal(文字常量)** 一个符号, 直接指出一个值, 如 12 指出整数值“十二”。
- loop(循环)** 重复执行的一段代码; 在 C++ 中, 通常是一条 for 语句或者一条 while 语句。
- mutable(可变的)** 可改变的; 与不可变的、常量、不变量相对。
- object(对象)** 1) 一个初始化过的已知类型的内存区域, 保存了该类型的一个值; 2) 一个内存区域。
- object code(目标代码)** 编译器的输出, 连接器的输入(连接器用来生成可执行程序)。
- object file(目标文件)** 包含目标代码的文件。
- object-oriented programming(面向对象程序设计)** 一种程序设计风格, 关注类和类层次的设计与使用。
- operation(操作)** 可执行某些动作的事物, 如函数或运算符。
- output(输出)** 计算生成的值(例如, 函数返回值或写到屏幕的一行字符)。
- overflow(溢出)** 生成的值无法存储目标。
- overload(重载)** 定义两个具有相同名字但参数(运算对象)类型不同的函数或运算符。
- override(覆盖)** 在派生类中定义一个函数, 其名字和参数类型都与基类中的一个虚函数完全相同, 从而通过基类的接口可以调用此函数。
- paradigm(范型)** 某种程度上有点自命不凡的称谓; 通常是暗示某个设计方案或程序设计风格优于其他。
- parameter(形参)** 对函数或模板的显式输入的声明。在函数被调用时, 可以通过形参的名字来访问传递来的实参。
- pointer(指针)** 1) 一个值, 用于识别内存中有类型的对象; 2) 保存这样一个值的变量。
- post-condition(后置条件)** 在退出一段代码(一个函数或一个循环)时必须成立的条件。
- pre-condition(前置条件)** 在进入一段代码(一个函数或一个循环)时必须成立的条件。
- program(程序)** 足够完整、可被计算机执行的代码(可能与其相关联的数据一起)。
- programming(程序设计)** 将问题解决方案表达为代码的艺术。
- programming language(程序设计语言)** 表达程序的语言。
- pseudo code(伪代码)** 用非正式表示方法, 而非程序设计语言描述的计算。
- pure virtual function(纯虚函数)** 在派生类中必须被覆盖的虚函数。
- Resource Acquisition Is Initialization, RAII(资源获取即初始化)** 一种基于作用域的资源管理基本技术。
- range(范围)** 值的序列, 可用起点和终点来描述。例如, [0: 5) 表示值 0、1、2、3 和 4。

regular expression(正则表达式) 模式的一种字符串表示法。

recursion(递归) 函数对自身的调用;参见迭代。

reference(引用) 1) 一个值,描述了内存中一个有类型值的位置;2) 保存这样一个值的变量。

requirement(要求) 1) 对于一个程序或一个程序片段的期望行为的描述;2) 描述了一个函数或一个模板根据其参数作出的假设。

resource(资源) 使用前必须申请,使用后应该释放的事物,如文件句柄、锁或内存。

rounding(舍入) 将一个值转换为数学上最接近的、低精度类型的值。

scope(作用域) 名字可以被访问的程序文本(源代码)区域。

sequence(序列) 可以按线性次序访问的一些值。

software(软件) 代码片段和关联的数据的集合;通常可以与程序互换使用。

source code(源代码) 程序员生成的代码,(理论上)其他程序员可读。

source file(源文件) 包含源代码的文件。

specification(规范) 描述了一段代码应该做什么。

standard(标准) 官方认可的某事物的定义,如程序设计语言。

state(状态) 一组值。

string(字符串) 字符序列。

style(风格) 一组程序设计技术,可保证一致地使用语言特性;有时使用其狭义含义:指代码的命名和外观形式的低层规则。

subtype(子类型) 派生的类型;具有另一个类型的所有属性,可能还有更多的属性。

supertype(超类型) 基类型;属性为另一个类型的属性的子集。

system(系统) 1) 一个程序或一组程序,目的是在一台计算机上执行某个任务;2) “操作系统”的简称,即计算机上的基本执行环境和工具。

template(模板) 由一个或多个类型或(编译时)值所参数化的类或函数;用来支持泛型程序设计的基本 C++ 语言特性。

testing(测试) 查找程序中错误的系统化方法。

trade-off(折衷) 在多种设计和实现标准之间的权衡。

truncation(截断) 从一个类型转换到另一个类型的过程中,由于目标状态无法准确表示要转换的值,从而丢失了信息。

type(类型) 定义了一组可能值及一组操作。

uninitialized(未初始化) 对象在初始化之前的(未定义)状态。

unit(单位/单元) 1) 标准度量,使得值具有实际意义(如千米,度量距离);2) 整体中区别于其他的(如命名的)部分。

use case(用例) 程序的特定(通常是简单)的使用方式,可以测试程序的功能、展示其目的。

value(值) 内存中的一组二进制位,按某种类型解释其意义。

variable(变量) 给定类型的命名对象;除非未初始化,否则包含一个值。

virtual function(虚函数) 可在派生类中覆盖的成员函数。

word(字) 计算机中内存的基本单元,通常对应一个整数。

参考书目

- Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools, Second Edition* (usually called "The Dragon Book"). Addison-Wesley, 2007. ISBN 0321547985.
- Andrews, Mike, and James A. Whittaker. *How to Break Software: Functional and Security Testing of Web Applications and Web Services*. Addison-Wesley, 2006. ISBN 0321369440.
- Austern, Matthew H. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley, 1999. ISBN 0201309564.
- Austern, Matt, ed. *Draft Technical Report on C++ Standard Library Extensions*. ISO/IEC PDTR 19768. www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1836.pdf.
- Bergin, Thomas J., and Richard G. Gibson, eds. *History of Programming Languages – Volume 2*. Addison-Wesley, 1996. ISBN 0201895021.
- Blanchette, Jasmin, and Mark Summerfield. *C++ GUI Programming with Qt 4*. Prentice Hall, 2006. ISBN 0131872493.
- Boost.org. "A Repository for Libraries Meant to Work Well with the C++ Standard Library." www.boost.org.
- Cox, Russ. "Regular Expression Matching Can Be Simple and Fast (but Is Slow in Java, Perl, PHP, Python, Ruby, . . .)." <http://swtch.com/~rsc/regexp/regexpl.html>.
- dmoz.org. <http://dmoz.org/Computers/Programming/Languages>.
- Freeman, T. L., and Chris Phillips. *Parallel Numerical Algorithms*. Prentice Hall, 1992. ISBN 0136515975.
- Gamma, Erich, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN 0201633612.
- Goldthwaite, Lois, ed. *Technical Report on C++ Performance*. ISO/IEC PDTR 18015. www.research.att.com/~bs/performanceTR.pdf.
- Gullberg, Jan. *Mathematics – From the Birth of Numbers*. W. W. Norton, 1996. ISBN 039304002X.
- Hailpern, Brent, and Barbara G. Ryder, eds. *Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages (HOPL-III)*. San Diego, CA, 2007. <http://portal.acm.org/toc.cfm?id=1238844>.
- Henricson, Mats, and Erik Nyquist. *Industrial Strength C++: Rules and Recommendations*. Prentice Hall, 1996. ISBN 0131209655.
- ISO/IEC 9899:1999. *Programming Languages – C*. The C standard.
- ISO/IEC 14882:2003. *Programming Languages – C++*. The C++ standard.
- Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, first edition, 1978; second edition, 1988. ISBN 0131103628.
- Knuth, Donald E. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Third Edition*. Addison-Wesley, 1998. ISBN 0201896842.
- Koenig, Andrew, ed. *The C++ Standard*. ISO/IEC 14882:2002. Wiley, 2003. ISBN 0470846747.
- Koenig, Andrew, and Barbara E. Moo. *Accelerated C++: Practical Programming by Example*. Addison-Wesley, 2000. ISBN 020170353X.
- Langer, Angelika, and Klaus Krefl. *Standard C++ IOStreams and Locales: Advanced Programmer's Guide and Reference*. Addison-Wesley, 2000. ISBN 0201183951.
- Lippman, Stanley B., Josée Lajoie, and Barbara E. Moo. *The C++ Primer*. Addison-Wesley, 2005. ISBN 0201721481. (Use only the 4th edition.)
- Lockheed Martin Corporation. "Joint Strike Fighter Air Vehicle Coding Standards for the System Development and Demonstration Program." Document Number 2RDU00001 Rev C. December 2005. Colloquially known as "JSF++." www.research.att.com/~bs/JSF-AV-rules.pdf.
- Lohr, Steve. *Go To: The Story of the Math Majors, Bridge Players, Engineers, Chess Wizards, Maverick Scientists and Iconoclasts – The Programmers Who Created the Software Revolution*. Basic Books, 2002. ISBN 9780465042265.
- Maddock, J. boost::regex documentation. www.boost.org and www.boost.org/doc/libs/1_36_0/libs/regex/doc/html/index.html.
- Meyers, Scott. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley, 2001. ISBN 0201749629.
- Meyers, Scott. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs, Third Edition*. Addison-Wesley, 2005. ISBN 0321334876.

- Musser, David R., Gillmer J. Derge, and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library, Second Edition*. Addison-Wesley, 2001. ISBN 0201379236.
- Programming Research. *High-integrity C++ Coding Standard Manual Version 2.4*. www.programmingresearch.com.
- Richards, Martin. *BCPL – The Language and Its Compiler*. Cambridge University Press, 1980. ISBN 0521219655.
- Ritchie, Dennis. "The Development of the C Programming Language." *Proceedings of the ACM History of Programming Languages Conference (HOPL-2)*. *ACM SIGPLAN Notices*, Vol. 28 No. 3, 1993.
- Salus, Peter. *A Quarter Century of UNIX*. Addison-Wesley, 1994. ISBN 0201547775.
- Sammet, Jean. *Programming Languages: History and Fundamentals*. Prentice Hall, 1969. ISBN 0137299885.
- Schmidt, Douglas C., and Stephen D. Huston. *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*. Addison-Wesley, 2002. ISBN 0201604647.
- Schmidt, Douglas C., and Stephen D. Huston. *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Addison-Wesley, 2003. ISBN 0201795256.
- Schwartz, Randal L., Tom Phoenix, and Brian D. Foy: *Learning Perl, Fourth Edition*. O'Reilly, 2005. ISBN 0596101058.
- Scott, Michael L. *Programming Language Pragmatics*. Morgan Kaufmann, 2000. ISBN 1558604421.
- Sebesta, Robert W. *Concepts of Programming Languages, Sixth Edition*. Addison-Wesley, 2003. ISBN 0321193628.
- Shepherd, Simon. "The Tiny Encryption Algorithm (TEA)." www.tayloredge.com/reference/Mathematics/TEA-XTEA.pdf and <http://143.53.36.235:8080/tea.htm>.
- Stepanov, Alexander. www.stepanovpapers.com.
- Stewart, G. W. *Matrix Algorithms, Volume I: Basic Decompositions*. SIAM, 1998. ISBN 0898714141.
- Stone, Debbie, Caroline Jarrett, Mark Woodroffe, and Shailey Minocha. *User Interface Design and Evaluation*. Morgan Kaufmann, 2005. ISBN 0120884364.
- Stroustrup, Bjarne. "A History of C++: 1979–1991." *Proceedings of the ACM History of Programming Languages Conference (HOPL-2)*. *ACM SIGPLAN Notices*, Vol. 28 No. 3, 1993.
- Stroustrup, Bjarne. *The Design and Evolution of C++*. Addison-Wesley, 1994. ISBN 0201543303.
- Stroustrup, Bjarne. "Learning Standard C++ as a New Language." *C/C++ Users Journal*, May 1999.
- Stroustrup, Bjarne. *The C++ Programming Language (Special Edition)*. Addison-Wesley, 2000. ISBN 0201700735.
- Stroustrup, Bjarne. "C and C++: Siblings"; "C and C++: A Case for Compatibility"; and "C and C++: Case Studies in Compatibility." *The C/C++ Users Journal*, July, Aug., and Sept. 2002.
- Stroustrup, Bjarne. "Evolving a Language in and for the Real World: C++ 1991–2006." *Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages (HOPL-III)*. San Diego, CA, 2007. <http://portal.acm.org/toc.cfm?id=1238844>.
- Stroustrup, Bjarne. Author's home page, www.research.att.com/~bs.
- Sutter, Herb. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley, 2000. ISBN 0201615622.
- Sutter, Herb, and Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison-Wesley, 2004. ISBN 0321113586.
- University of St. Andrews. The MacTutor History of Mathematics archive. <http://www-gap.dcs.st-and.ac.uk/~history>.
- Wexelblat, Richard L., ed. *History of Programming Languages*. Academic Press, 1981. ISBN 0127450408.
- Whittaker, James A. *How to Break Software: A Practical Guide to Testing*. Addison-Wesley, 2003. ISBN 0321194330.
- Wood, Alistair. *Introduction to Numerical Analysis*. Addison-Wesley, 1999. ISBN 020134291X.